A Spatially Partitioned Parallel Simulation of Colliding Objects

THESIS

Robert S. Moser
Captain, USAF

AFIT/GCS/ENG/GCS91D-15

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1991	Master's Thesis

4. TITLE AND SUBTITLE
A Spatially Partitined Parallel Simulation of Colliding Objects

5. FUNDING NUMBERS

6. AUTHOR(S)
Robert S. Moser, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/91D-15

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
ASAS Project Office, SFAE-CC-INT-OR, 1500 Planning Research Drive, Mclean VA 22102-5099

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This study investigated the application of a conservative synchronization paradigm to the classical, distributed pool balls simulation executed on an eight node, Intel iPSC/2 hypercube. Wieland's concept of spatial partitioning and limited data replication was used. Analysis has shown that 100% parallelization of execution is possible in a conservative environment via assignment of multiple sectors to nodes. Two conservative formulations for minimum safe time were derived. A tradeoff exists between scalability and efficiency. Optimum sectoring prediction has been shown possible through application of linear regression techniques. The results of this research reveal that a conservative approach to distributed, discrete event simulations can achieve significant speedup.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Discrete Event Simulation, Hypercube, Parallel Simulation	108
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

AFIT/GCS/ENG/GCS91D-15

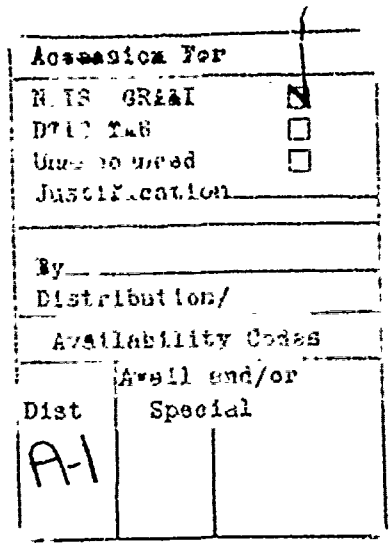
A Spatially Partitioned Parallel Simulation of Colliding Objects

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Computer Engineering

Robert S. Moser, B.S.E.E
Captain, USAF

December, 1991



Preface

The purpose of this study was to investigate the application of a conservative synchronization paradigm for parallel discrete event simulations; specifically, to solve the classical pool balls simulation. This thesis effort has demonstrated that the conservative approach can produce comparable performance to an optimistic approach by comparing the results of the pool balls simulations produced at Cal Tech with those of this thesis.

This thesis effort also demonstrates the viability of spatially partitioning a simulation model in a conservative environment. Several design approaches were analyzed and their respective advantages and disadvantages were derived. Two conservative formulations for a minimum safe time were developed both of which maintain system correctness and simulation progress. The tradeoff between them is shown to be scalability versus execution efficiency. The more efficient, less scalable one, was chosen for empirical study.

In the development of the simulation software, I had to acquaint myself with the C programming language. Were it not for the invaluable help of Lt Kevin Hanrahan, I would never have developed a working program. I would like to thank Professor Gary Lamont for his ceaseless encouragement and brilliant inspiration. Without his help, I would have pursued many wrong approaches in system design, theory and writing. He was particularly useful in the application of predicate logic for developing the theorems and proofs presented in this thesis.

I am deeply indebted to my wife Kelly and my daughter Rachel for their encouragement, support, patience and love. The time required to fulfill the requirements of Master of Science of Computer Engineering has been extraordinary. Without the direct support of my family, I could not have maintained the exacting pace demanded of my studies. Thank you.

Robert S. Moser

Table of Contents

List of Figures

List of Tables

AFIT/GCS/ENG/GCE91D-15

Abstract

This study investigated the application of a conservative synchronization paradigm to the classical, distributed pool balls simulation executed on an eight-node, Intel iPSC/2 hypercube. Wieland's concept of spatial partitioning and limited data replication was used. Analysis has shown that 100% parallelization of execution is possible in a conservative environment via assignment of multiple sectors to nodes. Two conservative formulations for minimum safe time were derived. A tradeoff exists between scalability and efficiency. Optimum sectoring prediction has been shown possible through application of linear regression techniques. The results of this research reveal that a conservative approach to distributed, discrete event simulations can achieve significant speedup.

A Spatially Partitioned Parallel Simulation of Colliding Objects

I. Introduction

This thesis investigates the application of a conservative synchronization paradigm for the execution of parallel discrete event simulations implementing spatially partitionable models. The methodology developed for this thesis is of particular interest to operations researchers and simulation system designers because a parallel processor having N nodes can potentially execute a distributed simulation up to N times faster than a single processor (21:315). The realization of this potential increase in performance is the primary motivation to distribute simulations over many processors. Furthermore, a distributed approach may be the only practicle or possible solution to some large, complex models. For example, Misra has investigated a sequential simulation of a complex telephone switch. Misra assumed that the switch can generate about 100 internal messages while completing a local telephone call and that 100 switches per second can be accommodated by a complex switch. A sequential simulation simulating 15 minutes of real time will generate nearly 10 million messages requiring several hours of simulation time on a very fast uniprocessor (18).

This thesis develops a general methodology from which many systems may be modeled in a distributed manner. This methodology is developed from the investigation of a specific distributed simulation in which pool balls move about on a pool table and collide with one another and with the pool table with perfect elasticity. There are several reasons why this simulation has been chosen for analysis.

1. The model is simple to comprehend, thereby emphasizing the *process* of developing a distributed simulation and not the simulation itself.
2. Basic simulation parameters such as computational loading, number of processors, number of logical processes, number of simulated objects, etc, are easily varied for performance measurement and analysis.

3. The pool balls simulation concept has become a classical simulation problem having been developed and conceived at Cal Tech under the title 'Colliding Pucks' (13) and benchmarked at the Jet Propulsion Laboratory under the title 'Pool Balls Benchmark' (4, 3). The pool balls simulation concept has also been studied using a modified version of the Time Warp operating system (17) and using a time driven simulation approach (7).
4. The published benchmark results from the Jet Propulsion Lab allow some performance comparisons to be made between the conservative and optimistic paradigm implementations for the pool balls simulation.

1.1 Background

According to Banks and Carson, 'a simulation is the imitation of the operation of a real-world process or system over time (2:2).' A *model* is a representation of a real-world system and takes the form of a set of assumptions concerning the behavior of the system. If the model of a real-world system accurately reflects the behavior of the system, then a simulation can be used to study the system without changing the real-world system. This form of experimentation can increase a user's knowledge of the system. Simulation can also be used to experiment with models of systems that do not yet exist, thereby providing an often used system design tool for complex and costly systems (2:4). Pritsker (19:6) states that simulations of real-world systems provide the experimenter with inferences about systems

> '... without building them, if they are only proposed systems; without disturbing them, if they are operating systems that are costly or unsafe to experiment with; without destroying them, if the object of an experiment is to determine their limits of stress.'

For the Department of Defense, military battle simulations precisely conform to all of Pritsker's observations. Wars are certainly unsafe to experiment with. During times of peace battle simulations provide valuable information concerning our preparedness for war. Many battle simulations are complex enough that sequential uniprocessors require several hours and even days to simulate a relatively short tactical scenario (10). There are still several questions regarding the parallelization of complex simulations. These questions include the following:

1. How should the problem domain be partitioned?
2. Which method of synchronization should be used?
3. Can load balancing be achieved across N processors?

This thesis investigates each of these questions in the context of the classical pool balls simulation.

1.2 Thesis Statement

A concurrently executing non-queueing model discrete event simulation can achieve near linear speedup implemented with a conservative synchronization paradigm incorporating spatial partitioning and limited data replication.

1.3 Scope

This thesis effort investigates the parallelization of a conservative discrete event simulation incorporating a classical, spatially partitionable model. The specific model chosen for investigation is the well defined pool balls problem. The conclusions are applicable to any spatially partitionable model such as battlefield simulations. The distributed software design is object oriented and spatially partitioned. The movement of each pool ball is recorded to disk for analysis and graphics display. The user can specify the number of pool balls to simulate, the number of logical processes, the number of physical processes, the simulation run time and the pool table dimensions. The user may also specify various options such as writing to disk, printing to screen, collecting statistics and checking for errors. Each option selected affects the execution time of the overall simulation.

Each pool ball is created during initialization. The parameters of position specified as $X - Y$ coordinates and velocity specified as $X - Y$ vectors are randomly generated using a machine independent pseudo random number generator developed by Law and Kelton (14).

The performance of this pool table simulation is compared to that of Cal Tech's 'Colliding Pucks' experiment to gain insight towards the desirability of the conservative synchronization paradigm over the optimistic synchronization paradigm. This is particularly important because the conservative paradigm has been shown to require only as much memory as its sequential counterpart while the optimistic paradigm requires large amounts of memory and has the potential to exhaust memory before simulation termination.

1.4 Research Objectives

The objectives of this thesis effort are:

1. To demonstrate by example that a spatially partitionable model discrete event simulation can be parallelized on distributed loosely coupled processors using a conservative synchronization paradigm.
2. To demonstrate that the parallelization of a spatially partitionable model discrete event simulation can achieve reasonable speedup.
3. To demonstrate that the conservative synchronization paradigm is comparable to an optimistic synchronization paradigm when applied to a spatially partitionable model parallel discrete event simulation.

1.5 Assumptions

Several assumptions were made in the analysis, design and development of the pool balls simulation. As a minimum, the following equipment and hardware specifications were assumed:

1. A distributed loosely coupled hypercube having eight or more nodes.
2. Monotonicity of message traffic between nodes is strictly maintained. This requirement states that if messages $(m_1, m_2, ..., m_n)$ occur at time $(t_1, t_2, ..., t_n)$ such that $0 \leq t_1 \leq t_2 \leq t_n$ then the target nodes will receive the messages in the same order.
3. Dynamic allocation and deallocation of memory is allowed.

1.6 General Approach

The general approach for this thesis consisted of six steps:

1. A literature search was conducted. Familiarity with different types and classes of simulations was acquired and an understanding of the Chandy-Misra paradigm developed.
2. The requirements analysis was performed for the software system. The assumptions generated were required to conform to those developed by Cal Tech to provide a means for performance comparison.

3. The requirements analysis was validated and the pool balls simulation was designed. An object oriented approach was used to enhance software documentation, maintenance and changeability.

4. The software was written in a three-step process. First, a sequential pool balls simulation was designed implementing a non-partitioned pool table. Second, the sequential system from the first stage was modified so that the pool table could be partitioned into vertical 'slices'. Third, the partitioned pool table resulting from the second stage was parallelized on the Intel iPSC/2 hypercube. All three software versions were coded in the C programming language.

5. Various test simulations were developed which provided the speedup estimates and performance comparisons between Cal Tech's experiments and AFIT's experiments. Tests were executed using all three software stages to demonstrate output consistency.

1.7 Summary

A spatially partitionable model discrete event simulation can be parallelized onto a distributed, loosely coupled processor. Near linear speedup is achievable using a conservative synchronization paradigm. These assertions are demonstrated by implementing the well documented, classical pool balls simulation which was conceived and developed by scientists at Cal Tech and later benchmarked at the Jet Propulsion Laboratory. A conservative paradigm results in comparable performance to the Time Warp optimistic paradigm based upon the reported results from Cal Tech and the Jet Propulsion Lab.

II. Issues in Distributed Discrete Event Simulation

2.1 Introduction

This chapter surveys current literature on topics related to this thesis. This review is limited to on-going research in parallel discrete event simulations and briefly discusses various classes of computer architecture used in distributed processing.

2.2 Motivation for Parallel Computing

If a computer performs one instruction at a time in sequential fashion then the only possibility for increasing execution performance is to increase the speed at which instructions are performed. Despite the fact that VLSI technology has been doubling the performance of computing hardware every couple of years, it is doubtful that this trend can continue beyond the 21st century and is certain not to continue indefinitely (9:23).

An alternative approach for increasing execution performance is to design and use computer architectures that perform multiple instructions simultaneously. If a sequential processor requires T_s time to complete a process, then a parallel processor having M processors requires a lower bound of $\frac{T_s}{M}$ time to complete the same process, provided that each of the M processors is equal in power to the sequential processor and that all M processors are 100 percent utilized. The increase in run time performance is a factor less than or equal to M. This theoretical upper bound on parallel computing performance is the primary motivation towards parallel computing. In the future, this may be the only means available to decrease execution time. Even if technology can continue increasing the speed of sequential processors, the rate of performance increase is significantly less than the potential of establishing an M-fold increase by using massive parallelism.

2.3 Flynn's Taxonomy

Michael J. Flynn classified all digital computers into four categories according to the types of instruction and data streams used. An instruction stream is a sequence of instructions executed by a given computer. A data stream is a sequence of data representing input, output or temporary results used to calculate the output. Flynn's four categories are (12:32):

1. Single Instruction stream, Single Data stream (SISD).

2. Single Instruction stream, Multiple Data stream (SIMD).

3. Multiple Instruction stream, Single Data Stream (MISD).

4. Multiple Instruction stream, Multiple Data stream (MIMD).

The simplest architecture is the SISD class. These computers execute one instruction at a time and operate on one piece of data at a time. The most complex architecture is the MIMD class. MIMD computers execute multiple instructions simultaneously on different data. One may think of a MIMD computer as several processors tied together, each processor of which is a fully functional and often times powerful computer. The manner in which the processors of a MIMD architecture are tied together distinguishes loosely coupled and tightly coupled MIMD computers.

2.4 Tightly Coupled VS Loosely Coupled MIMD Architectures

The individual processors of a parallel processor architecture must cooperate with one another in order to solve a particular application. This cooperation often entails the sharing of data structures and variables which reside in memory. One approach to parallel architecture design is to have a global memory which each processor may access. This shared memory design is referred to as *tightly coupled.* This design has the advantage of internodal communications at memory speeds. The disadvantages include bus contention, cache coherence and memory access. Memory can only be read from or written to one address at a time; hence, the individual processors of a tightly coupled architecture often 'fight' over access rights to memory. Current bus-architecture technology limits the number of processors to 40 or 50. Cache coherence is a problem in that multiple processors may alter variables in memory even though some or all of the variables reside within an individual processor's cache. This poses the problem of having multiple variable values within the local caches of different processors (9:19).

An alternative approach to parallel architecture design is to have separate local memories owned and controlled by the individual processors. Such a design is referred to as *loosely coupled.* These designs prohibit processors from accessing memory variables outside of local memory. These variables must then be communicated via message passing which is considerably slower than the memory speeds achieved by the shared memory concept discussed above, but cache coherence and bus contention are not problems and the size of the architecture is scalable to several thousand processors depending upon the connectivity between the processors (9:20,21). Research continues as to the applicability

of both architecture designs towards specific problems. This thesis effort focuses on the application of the Intel iPSC/2 hypercube loosely coupled MIMD architecture towards the classical pool balls simulation.

2.5 Hypercube MIMD Architectures

A loosely coupled distributed architecture must send and receive communicated variables across interconnecting communications networks which are significantly slower than CPU cycles, bus cycles or even memory cycles. The approach toward keeping communications time to a minimum is to have each processor (often times referred to as a 'node') directly connected to every other processor so that the communications line is both short and direct. This type of fully connected MIMD architecture is known as a crossbar. Crossbars require M^2 links between M nodes which is expensive both in terms of hardware cost and size. Scalability of crossbars is severely restricted since the communications links grow with the square of M. Thus, the communications time complexity is $O(1)$ at the cost of $O(M^2)$ links (9:114-116). An architecture which uses fewer communications links has greater scalability but greater communications time. The least number of links possible between nodes is two represented by linear arrays and ring networks. These interconnection networks have communications time complexity of $O(M)$ at the cost of only $O(1)$ links (9:114-116). These types of networks work well if the application requires nodes to share data between themselves and their immediate neighbor.

The hypercube enjoys the greatest popularity amongst loosely coupled MIMD architectures because of its versatility, scalability and communications time. A hypercube has $M = 2^m$ processors interconnected as a binary cube. Each processor is a fully self contained computer with its own clock, CPU and local memory. Each processor also has m connections with other processors in the cube. Hence, the worst case communications time between any two processors is $O(\log M)$. This places the communications time between the high speed of the fully connected MIMD architecture and the ring architecture while preserving the capability of scalability. The Intel iPSC/2 hypercube has a front end processor that is directly connected to each node of the cube via a 10 Mbps ethernet connection. Each node employs a Direct Connect Module (DCM) which frees a node's CPU from directing message traffic. Each of the nodes is made up of a standard Intel 80386 processor rated at 4 MIPS (9:441-451).

2.6 *Performance Measures*

There are several measures of performance for parallel computing. The most common measurement is *speedup* which Hayes describes as the ratio of the total execution time on a sequential computer to the corresponding execution time on a parallel computer using M processors (11). Mathematically,

$$S(M) = \frac{T_s}{T_p} \tag{1}$$

Since $T_p \geq \frac{T_s}{M}$, the speedup $S \leq M$. Stone feels that the definition used by Hayes and others leads to ambiguity because the definition provides for inflated values. Instead, Stone states that speedup is the ratio of *the best possible* serial algorithm implementation to the parallel implementation (21:141). A speedup which measures the performance of the same algorithm implemented serially and in parallel should, according to Stone, be defined as *relative* speedup. This thesis uses Stone's concept of relative speedup for performance analysis.

Another useful performance measure is *efficiency* which Hayes describes as speedup per degree of parallelism (11:583), defined mathematically as:

$$E(M) = \frac{S(M)}{M} \tag{2}$$

2.7 *Taxonomy of Simulations and Simulation Models*

A simulation may be either discrete or continuous. A discrete system allows the state variables to change only at discrete points in time whereas a continuous system allows the state variables to change continously over time. A model is defined as a representation of a system. A model need only include the aspects of a real system under observation whose behavioral characteristics are intended for study. The model is hence a representation of a real world entity but it is also a simplification of that entity. Models are typically described by three attributes: static or dynamic, deterministic or stochastic, and discrete or continous. A static model represents an entity at a particular point in time (usually referred to as a Monte Carlo model) whereas a dynamic model represents an entity as it changes over time. The pool balls simulation uses a *dynamic* model. A deterministic model has a known set of inputs and results in a unique set of outputs. There are no random variables in a deterministic model. A stochastic model is probablistic and relies upon one

or more random variables as inputs. Due to the randomness of the inputs, a stochastic model must be considered only as an estimator of an entity's behavior. Statistical estimates sought as outputs of a stochastic model include the mean time between failure, the mean service, or mean wait time. The pool balls simulation uses a *stochastic* model as the pool balls are generated with random positions and velocities. A simulation model may be either discrete or continuous. A discrete simulation model represents an entity that changes only at discrete points in time. A continuous model represents an entity that changes constantly over time. Most queueing models are discrete. The pool balls model is *continous.* It is important to note that a continous model may be observed only at discrete points in time and a discrete model may be continously observed over time. The pool balls simulation is discrete but the model is continous as each pool ball continously changes with time (2:3-12).

A simulation may be time driven or event driven. A time driven simulation updates a dynamic simulation model by constant time intervals. With regard to the pool balls simulation, a time driven implementation would move each pool ball by a predetermined delta t. A time driven simulation may be allowed to process faster by increasing the delta t value thereby requiring fewer updates over a specified time interval; however, resolution of the simulation output decreases as the delta t increases.

An event driven simulation updates objects within a simulation model at discrete points in time which have been defined as 'events of interest'. If the events can be properly defined, the event driven simulation promises theoretical improvement over its time driven counterpart. This potential performance gain of the event driven approach arises from only having to calculate the state information for the exact set of events of interest. The time driven approach will calculate the state information for not only the set of events of interest but also for all of the incremental states corresponding to the delta times (which are not events of interest).

2.8 Distributed Discrete Event Simulation Paradigms

Chandy and Misra developed a conservative synchronization paradigm in 1979 for the successful implementation of a distributed discrete event simulation. Jefferson et al presented their optimistic synchronization paradigm in 1985. To date, all other proposed paradigms are variations and extensions to the two original paradigms. The problem that both paradigms overcome is the handling of out-of-sequence messages. Consider the pool balls scenario presented in Figure 1.

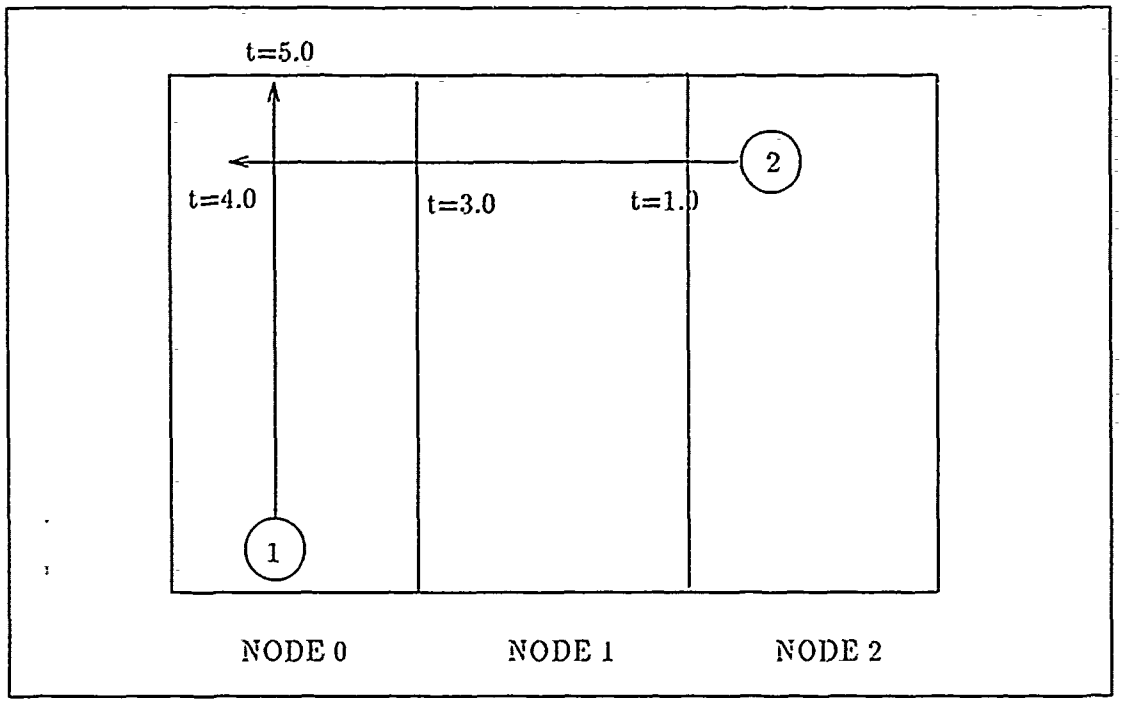


Figure 1. A Partitioned Pool Table with a Future Collision

In Figure 1, suppose that pool balls 1 and 2 will collide with one another at time $t = 4$. Let us assume that for the parallel implementation, the pool table is sliced into sectors with one sector allocated to each of three nodes. Each node knows only of the existence of its pool balls. The leftmost sector on node 0 cannot 'see' ball 2 on node 2 and vice versa. This situation causes node 0 to predict that ball 1 will strike the top horizontal cushion at time $t = 5$ while node 2 will predict that ball 2 will exit the sector at time $t = 1$. If both node 0 and node 2 execute their events simultaneously, node 2 will be correct and node 0 will be incorrect. Eventually, ball 2 will migrate to node 0 but the collision between the two pool balls will no longer be possible because ball 1 has already been simulated past the collision time of $t = 4.0$. The arrival of ball 2 at node 0 at time $= 3.0$ is an *out-of-sequence message* if node 0 has already simulated *past* time $t = 3.0$.

An optimistic strategy assumes that out-of-sequence messages will not occur; thus, every node in the distributed system processes all of the data that it can. As each event is processed, the state data is stored in memory. If an out-of-sequence message does occur as it would in the scenario of Figure 1, the node that receives the message reverses (referred to as rollback) the simulation back to the last event executed just prior to the out-of-sequence message. The simulation is then recalculated with the newly arrived message data.

A conservative strategy prevents out-of-sequence messages from occuring by preventing processors from executing until such a time that they can safely *guarantee* that no out-of-sequence messages will arrive. Thus, node 2 of Figure1 would have been allowed to process the exit event of ball 2, but node 0 would have been required to wait (i.e. to sit idle) until it was safe to process. The mechanism used by Chandy and Misra to guarantee system correctness is the Minimum Safe Time (MST) which is a calculated value of simulation time which guarantees that no out-of-sequence messages will occur up to time $t \leq MST$.

Both paradigms have strengths and weaknesses. The optimistic strategy requires large amounts of memory for rollback and can exhaust the memory during the simulation. Chandy and Misra have shown that their conservative paradigm requires only a bounded amount of memory and does not require more memory than a sequential simulation (15). Lipton and Mizell assert that Time Warp outperforms Chandy-Misra by a factor of p in the best case and cannot lag arbitrarily far behind Chandy-Misra in the worst case (16). This is based on the intuitive premise that Time Warp can 'win big' if it correctly guesses the correct choices concerning what events to process and what events not to process. Furthermore, even if a processor incorrectly processes an event, as in Figure 1, it is the processor which has processed furthest in simulation time which is penalized; therefore, the simulation is no slower than the slowest processor plus some constant overhead factor to enforce the roll back. Lin and Lazowska have taken a more analytical view but conclude basically the same thing. Their conclusion is based upon models of the Time Warp and Chandy-Misra paradigms which employ several underlying assumptions. Assumption 2.1 in Lin and Lazowska's paper states that each logical process is assigned to a dedicated processor. This assumption reduced the potential speedup of their model because not all conservative models require a one to one mapping between logical processes and processors. Let the number of logical processes be k and the number of processors be n. The probability of an idle processor using Chandy-Misra decreases as n decreases such that $k > n$ (15). Therefore, Lin and Lazowska's conclusions may be erroneous for conservative strategies that can assign multiple processes to processors. Lin and Lazowska referenced this fact in their concluding remarks. Assumption 2.2 in Lin and Lazowska's paper states that Time Warp can rollback a simulation in negligable time. This underlying assumption is perhaps required to reduce the variables in an analytical model, but the assumption is not realistic. Indeed, Lin and Lazowska state in their concluding remarks that the overhead of the Time Warp operations is greater than that of the Chandy-Misra operation. This discrepency has

been taken into consideration by Lipton and Mizell causing them to conclude that Time Warp is always within a constant factor of optimal. The question addressed by this thesis is whether or not a conservative approach can also be made within a factor of optimal and whether or not this factor can be higher than Time Warp. This thesis also demonstrates that the pool balls simulation can be effectively partitioned such that multiple logical processes can be assigned to each processor.

2.9 The Theories of Chandy and Misra

Each process in a physical system is simulated by a separate logical process. Chandy and Misra use the term LP for logical processes and PP for physical processes. The logic of an LP depends solely upon the PP that it is simulating. An LP_i has a communications link to LP_j if and only if PP_i has a communications (dependency) link to PP_j. All messages between LP_i and LP_j consist of a tuple (t,m) such that t represents the time of the message and m represents the contents of the message. An LP can only process up to the time of the latest tuple which was received. This condition is sufficient to guarantee that no out-of-sequence messages will be received by any LP and simulation correctness is thus guaranteed (3). These concepts form the basis for Chandy and Misra's original publication in 1979 subject to the following constraints:

1. A process may decide to send a message at any arbitrary time $t > 0$ (6:440).
2. For all message tuples of a simulation time period $(0, Z)$, $0 < t_1 < \cdots < t_k \leq Z$ (6:442).
3. A message is sent from LP_i to LP_j if and only if LP_i is ready to send the message and LP_j is ready to receive it (6:443).

The third constraint stated above allows for the possibility of deadlock. Chandy and Misra assert that all distributed discrete event simulations using a conservative paradigm are subject to deadlocks and therefore require a mechanism to accommodate it. Chandy and Misra provide three such mechanisms. The first two are straight forward deadlock detection and recovery and deadlock avoidance. The third mechanism which is both favored and pioneered by Chandy and Misra is the concept of NULL messages. Such a message consists of the tuple $(t, NULL)$ which does not exist in the physical system. The presence of a NULL message allows LP's to continue processing up to the time of the NULL message when they would otherwise be blocked. The following queueing network taken from Chandy and Misra's article serves to demonstrate this process (6:446).

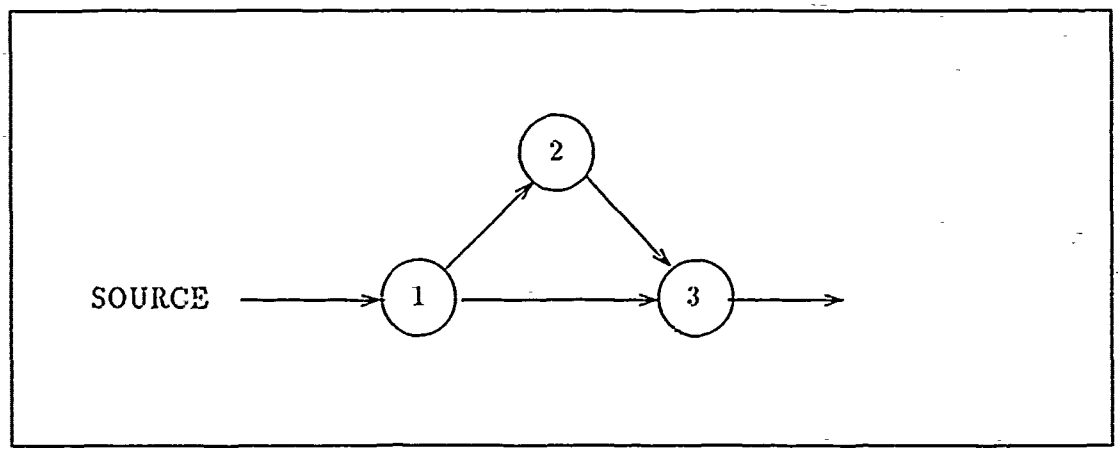


Figure 2. A Queuing Network that can Deadlock

1. Source outputs $(50, m_1)$ to LP1.
2. LP1 outputs $(50, m_1)$ to LP2.
3. LP2 outputs $(55, m_1)$ to LP3. At this point, LP3 is still waiting to receive a message from LP1.
4. Source outputs $(100, m_2)$ to LP1.
5. LP1 outputs $(100, m_2)$ to LP2.
6. LP2 *waits* to output $(105, m_2)$ to LP3 because LP3 is not ready to receive another message from LP2 until it first receives a message from LP1.
7. Source outputs $(150, m_3)$ to LP1.
8. LP1 *waits* to output $(150, m_3)$ to LP2 because LP2 is not ready to receive another message from LP1 until $(105, m2)$ is sent to LP3.

At this point, the queueing network is *deadlocked* because LP3 is expecting a message from LP1 which it never received; therefore, LP3 cannot accept LP2's second message which cannot accept LP1's third message. LP1 will never be able to send LP3 a message because it is waiting for LP2 to accept LP1's third message. Every LP is thus waiting upon every other LP.

Insertion of NULL messages in Chandy and Misra's example avoids the possibility of deadlock. At the time of the arrival of the first message m_1 at LP1, LP1 determined the message should be addressed to LP2. Even though this message was not the type required by LP3, LP1 can still send a NULL message to LP3 at time $t = 50$. This

would have allowed LP2 to send the tuple $(105, m_2)$ to LP3 and LP3 could have then received the tuple. It should be clear that the NULL messages corresponding to the tuples $(50, NULL)$, $(100, NULL)$ and $(150, NULL)$ are sufficient to avoid the deadlock situation. The drawback to the NULL message approach is that logical processes are required to process more messages than exist in the physical system. Such approaches are ill suited towards course grain machines due to the excessive message traffic which can result.

Chandy and Misra expanded their theory and presented a follow up paper in 1981 (5). They developed a new constraint such that within a physical system, the '...behavior of a PP at time t cannot be influenced by messages transmitted to it *after* t (5:198)'. This necessary condition is called the *realizability condition.* This leads indirectly to the assertion that if LP_i sends LP_j a message (t_k, m_k), it implies that all messages from PP_i to PP_j have been simulated up to time t_k (5:199).

2.10 Event Modeling

Schruben defines a system as a set of entities. Entities may fall into one of two general categories referred to as *resident* and *transient.* A resident entity is considered to have the property of permanent existence. For example, a simulation of a factory might model the machines in the factory as resident entities since the machines are always there. A transient entity is not permanent. Thus the factory simulation might instead model the behavior of the parts as they pass from machine to machine (20:101-102). With respect to the pool balls simulation, this corresponds to modeling the pool table sectors as resident entities or modeling the behavior of the pool balls as transient entities. Schruben maintains that both viewpoints are equally valid and both viewpoints should be considered during the simulation design phase.

2.11 Spatial Partitioning

Wieland and Hawley researched the application of sectoring a battlefield for the STB89 tactical battle simulation (22). Each object in the simulation has a 'perception radius' which defines the range that an object can detect another object. As an object approaches a partition border, the perception radius eventually becomes tangent to the border. If the object continues to move toward the border, the perception radius will protrude into the adjacent sector. This condition requires that the object have knowledge of

all objects residing within the adjacent sector. Wieland identified two techniques available to accommodate this condition. The first is to have the sector owning the moving object receive a copy of all of the objects residing within the adjacent sector. This technique is presumed to yield poor results because the search space of the original sector will approach the search space of a non-partitioned battlefield thereby negating any potential gains. The second approach that Wieland identified is to provide the second sector with a copy of the object which is moving towards it. Hence, only one object is passed and the search space of both sectors has at most an $O(1)$ increase. Wieland's strategy identifies three critical events as shown in Figure 3 (22:3).

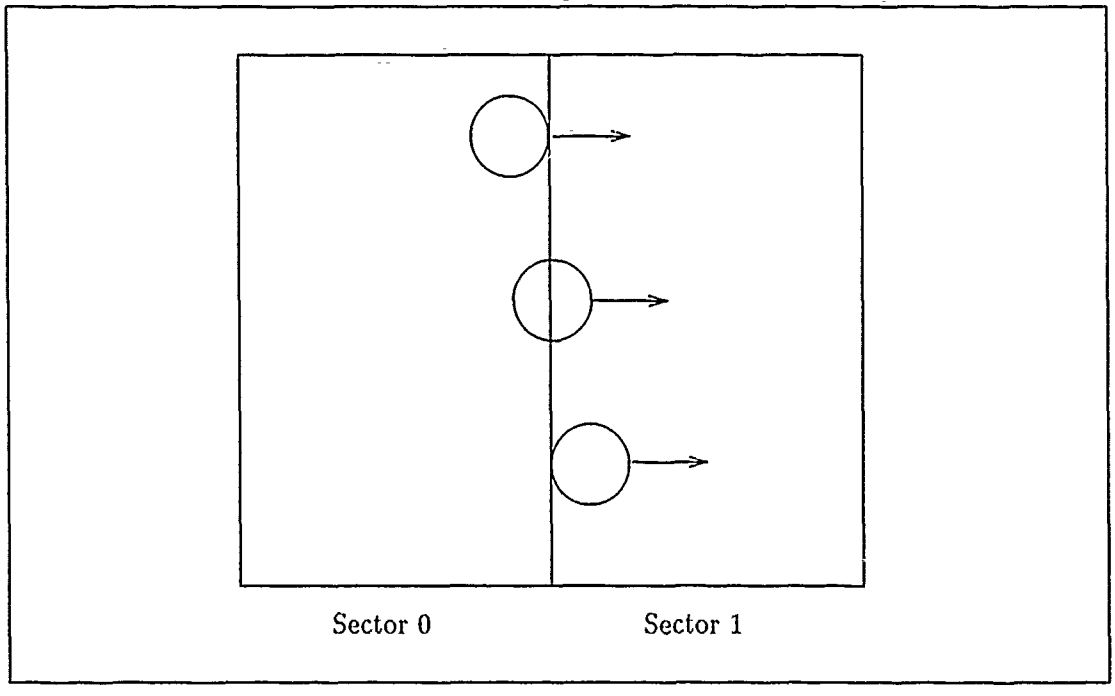


Figure 3. Wieland's Grid to Grid Proximity Detection

Wieland states that the first event occurs whenever a part of an object's perception radius is tangent to the sector boundary. At this time, an 'Add_Unit' message is sent to the adjacent sector; however, the adjacent sector does not 'control' the object added. Wieland refers to this additional object message as 'data replication' since the object exists on two processes (sectors). The second event occurs when the object's center (i.e. that which defines the object's location) crosses the sector border. At this time, a change of ownership message is sent from the original sector to the gaining sector. The third event

occurs when the object's perception radius is again tangent to the sector border. At this time, a 'Delete_Unit' message is sent from the gaining sector to the losing sector.

Wieland added a comment in his analysis stating that the second event which he identified in his data partitioning and replication strategy could be eliminated. There were no comments regarding any experimental studies concerning this latter assertion.

III. Requirements Analysis

3.1 Introduction

The Air Force Institute of Technology is interested in developing techniques for the successful design and implementation of parallel discrete event simulations beneficial to the Department of Defense. Two specific applications include hardware design simulations (using VHDL, for example) and battlefield simulations. The Institute is currently emphasizing distributed simulations incorporating the conservative synchronization strategy rather than the optimistic strategy. Unfortunately, there has been little reported in the literature on either design or implementation for non-queueing theoretic models for discrete event simulations which use the conservative paradigm. Several questions need to be addressed before attacking the parallelization of large software systems such as the battlefield simulations used by the DoD. These questions include:

1. How can (or should) the problem domain be partitioned?
2. Can a distributed simulation achieve comparable (or superior) performance using a conservative strategy rather than an optimistic strategy?
3. Can speedup be achieved to a large enough degree to make the parallelizing of existing DoD simulations worthwhile?
4. Are successful distributed simulations incorporating a conservative strategy scalable?

Insights into the above questions may be found through the design and implementation of a small scale simulation. The classical pool balls simulation is ideal subject matter because it has many of the same processes that a battlefield simulation has. These processes include the handling of moving objects through space (albeit two dimensional), search algorithms for object event identification, geographic domain structure and application of spatial partitioning with limited data replication. Furthermore, the pool balls problem domain is well understood and documented and test results are available for the comparison between an optimistic strategy and a conservative one. This will also be the first research effort at the Institute for the design and implementation of a non-queueing problem incorporating an event list with the conservative paradigm; therefore, this research will provide valuable experience to the Institute for follow on work.

3.2 Requirements

The following requirements were established for this project.

1. **Comply with Stated Requirements of Cal Tech's Pool Ball Experiment.** It was desirable to maintain consistency with Cal Tech's simulation for all matters of relevance so that a comparison could be made between the two paradigms used (3). Some of the stated requirements of the Cal Tech experiment were not considered relevant and were thus ignored. Two specific instances include Cal Tech's requirement that the pool balls have variable radius and mass. The following requirements were extracted from Cal Tech's requirements.

   (a) Each pool ball has measurable size and measurable mass (i.e. the pool balls are not point particles) .

   (b) Collisions between pool balls are perfectly elastic thereby conserving energy and momentum.

   (c) The pool balls move without friction.

   (d) Rotational energy of pool balls is ignored.

   (e) The enforcement of collisions follows the physical properties of elastic collisions (i.e. the collisions are realistic).

   (f) The pool table has no 'pockets'; therefore, the number of pool balls for any given simulation does not change for the duration of the simulation.

   (g) Every pool ball occupies a unique space on the table and no two balls can occupy a portion of the same space (i.e. overlap is not allowed).

2. **The Simulation Will Support Variable Quantities of Pool Balls.** The upper bound on the number of pool balls is specified in terms of the memory available for dynamic allocation of pool ball instantiations and what will physically fit on the pool table.

3. **The Dimensions of the Pool Table Will Be Modifiable.** Although the length and width of the pool table is not deemed a highly dynamic variable, the capability for changing the length and width is required. This factor allows for changing densities of pool balls on the table as well as the ability to expand the pool table so that more pool balls can physically be located on it.

4. **The Radius and Mass of Each Pool Ball Will Be Equal.**
5. **This Simulation Will Incorporate a Conservative Synchronization Strategy.**
6. **The AFIT Generic Simulation Shell Will be Used Both in the Simulation Design and Implementation.** AFIT has developed a generic simulation driver for discrete event simulations for standardization purposes. This driver is object oriented and consists of a next event queue, a clock, an event manager, and a simulation controller that sequences through the simulation. The application specific software interfaces to AFIT's simulation driver.

3.3 Developing the Equations of Motion

The requirements for this thesis demand that the collisions between pool balls conform to the principles of elastic collision and frictionless motion. Since the development of the equations of motion was not specifically stated in previous literature, it is shown here to support this research. The initial equations used can be found in most elementary physics books.

The equations for conservation of energy and momentum for two pool balls B and b having initial velocity vectors V_0 and v_0 and final velocity vectors V_1 and v_1 are respectively as follows:

$$\frac{1}{2}m|\vec{v}_0|^2 + \frac{1}{2}m|\vec{V}_0|^2 = \frac{1}{2}m|\vec{v}_1|^2 + \frac{1}{2}m|\vec{V}_1|^2 \quad (3)$$

$$m\vec{v}_o + m\vec{V}_0 = m\vec{v}_1 + m\vec{V}_1 \quad (4)$$

Equations (3) and (4) form the basis to develop algorithms for solving the events of pool balls colliding with cushions and pool balls colliding with one another.

Another useful equation is that of frictionless motion on a two dimensional plane.

$$\begin{bmatrix} X_1 \\ Y_1 \end{bmatrix} = \begin{bmatrix} X_0 + V_x * \Delta T \\ Y_0 + V_y * \Delta T \end{bmatrix} \quad (5)$$

The set of events S contains five event types which have been defined as events of interest. These events are $S = \{VERT, HOR, COLL, PART, EXIT\}$. These event types

correspond to vertical cushion collisions, horizontal cushion collisions, collisions between pool balls, reaching a sector boundary and crossing a sector boundary respectively. The partition and exit events are only used for a partitioned pool table in accordance with Wieland's two step sectoring strategy (22). The following sections derive the equations used to calculate the time at which each of the five event types will occur and the equations used to calcute the ball state information for a pool ball that executed an event type. It is assumed that the pool table is a rectangle with its top and bottom cushions parallel to the X-axis and its left and right cushions parallel to the Y-axis. Throughout this thesis, the top and bottom cushions are referred to as 'horizontal cushions' while the left and right cushions are referred to as 'vertical cushions'. Figure 4 shows the layout of the pool table on an X-Y coordinate system. Each ball has its position defined by the X and Y coordinates of the ball's center. Each pool ball has the following state information:

- ball time tag
- X
- Y
- V_x
- V_y

The pool balls algorithm calculates the time of the next event for each pool ball. Each pool ball will have an event corresponding to one of the event types in S. To determine which of the event types will occur for any given pool ball, an event time is calculated for each of the five event types. By definition of monotonicity, the earliest calculated event time for the set S defines the next possible event for a pool ball. The time t_2 of each event type in S is determined by $t_2 = t_1 + \Delta T$ where ΔT is calculated using equation 5 and t_1 is the current ball time tag.

3.3.1 Event Calculations for Collisions with Cushions This section develops the equations used to calculate the time at which a pool ball will strike any of the four table cushions and the ball state information after striking any of the four table cushions.

3.3.1.1 Calculating ΔT to Strike a Cushion The X-axis coordinate is known for both the left and right vertical cushions. These values are 0.00 and X_{table_length} respectively where X_{table_length} is the user defined length of the pool table. The velocity V_x of

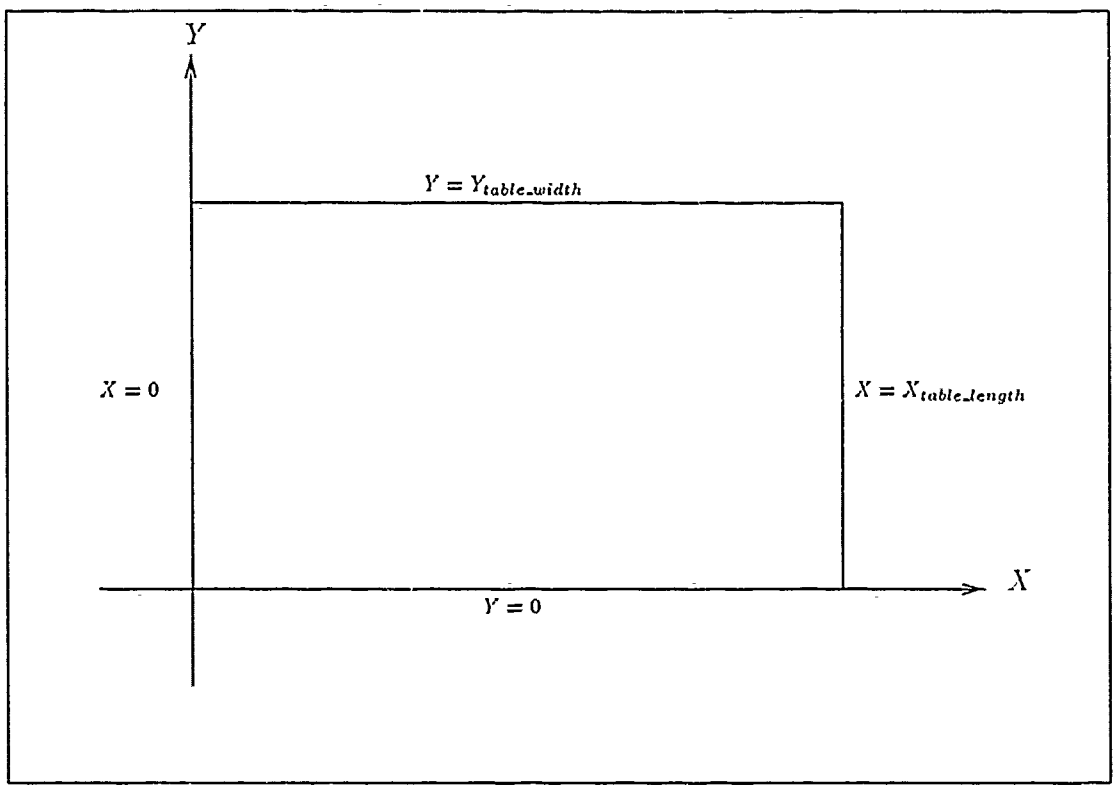


Figure 4. Layout of a Pool Table on an X/Y Axis System

a pool ball determines whether the ball will strike the left or right vertical cushion. The known X-axis coordinate for the appropriate vertical cushion is substituted into equation 5 as X_1. Using equation 5, the delta time of the collision is defined by

$$\Delta T = \frac{(X_1 - X_0)}{V_x}$$

The Y-axis coordinate is known for both top and bottom horizontal cushions. These values are 0.00 and Y_{table_width} respectively where Y_{table_width} is the user defined width of the pool table. The velocity V_y of a pool ball determines whether the ball will strike the top or bottom horizontal cushion. The known Y-axis coordinate for the appropriate horizontal cushion is substitued into equation 5 as Y_1. Using equation 5, the delta time of the collision is defined by

$$\Delta T = \frac{(Y_1 - Y_0)}{V_y}$$

3.3.1.2 Calculating the State Information for Cushion Collisions The ball time tag is simply replaced with the time of the event to be executed. This time is stored as a parameter with the event message. The parameters X and Y can be calculated directly from equation 5. The velocities v_x and v_y are calculated indirectly from equations 3 and 4. Let $\vec{V}$ be the velocity of the cushion and $\vec{v}$ be the velocity of the ball object. The mass of the cushion is much greater than the mass of the ball; therefore, the following equations can be used for a vertical cushion event.

$$\begin{bmatrix} v_{x1} \\ v_{y1} \end{bmatrix} = \begin{bmatrix} -v_{x0} \\ v_{y0} \end{bmatrix} \tag{6}$$

If the event type is a horizontal cushion event, then equation 7 defines the new values of v_x and v_y.

$$\begin{bmatrix} v_{x1} \\ v_{y1} \end{bmatrix} = \begin{bmatrix} v_{x0} \\ -v_{y0} \end{bmatrix} \tag{7}$$

Equation 6 requires the X-axis velocity to change direction while the Y-axis velocity remains unchanged. Equation 7 is just the opposite.

3.3.2 Calculations for Partition and Exit Events This section develops the equations used to calculate the time at which a pool ball will reach a sector border (partition event), or depart a sector to an adjacent one (exit event). This section also develops the equations used to update the ball state information after a partition or exit event.

3.3.2.1 Calculating ΔT for Partition and Exit Events The X-axis coordinate is known for both the left and right borders of any interior sector. These values are determined dynamically during initialization based upon the user specified table dimensions and the number of sectors desired. For partition events, a pool ball is moved to a location corresponding to $(X_{left\_border} + R)$ or $(X_{right\_border} - R)$ where R is the specified pool ball radius. For exit events, a pool ball crosses a partition into the neighboring sector and moves a distance of $2R$. The new X-axis coordinates will be $(X_{left\_border} - R)$ or $(X_{right\_border} + R)$. The velocity V_x of a pool ball determines whether the ball will reach the left or right border, or exit the sector to the left or to the right. Using equation 5, the delta time of the partition or exit event is defined by

$$\Delta T = \frac{(X_1 - X_0)}{V_x}$$

3.3.2.2 Calculating the State Information The ball time tag is simply replaced with the time of the event to be executed. The parameters X and Y can be calculated directly from equation 5. Since partition and exit events are not associated with collisions, the pool ball velocities V_x and V_y do not change.

3.3.3 Event Calculations for Collisions Between Pool Balls To determine if a pool ball will strike another pool ball, the two point formula for the distance between two lines can be used. This formula is stated as equation (8).

$$l^2 = (X_1 - x_1)^2 + (Y_1 - y_1)^2 \tag{8}$$

where l is the straight line distance between the centers of the two colliding pool balls at the point of impact. This is shown in Figure 5.

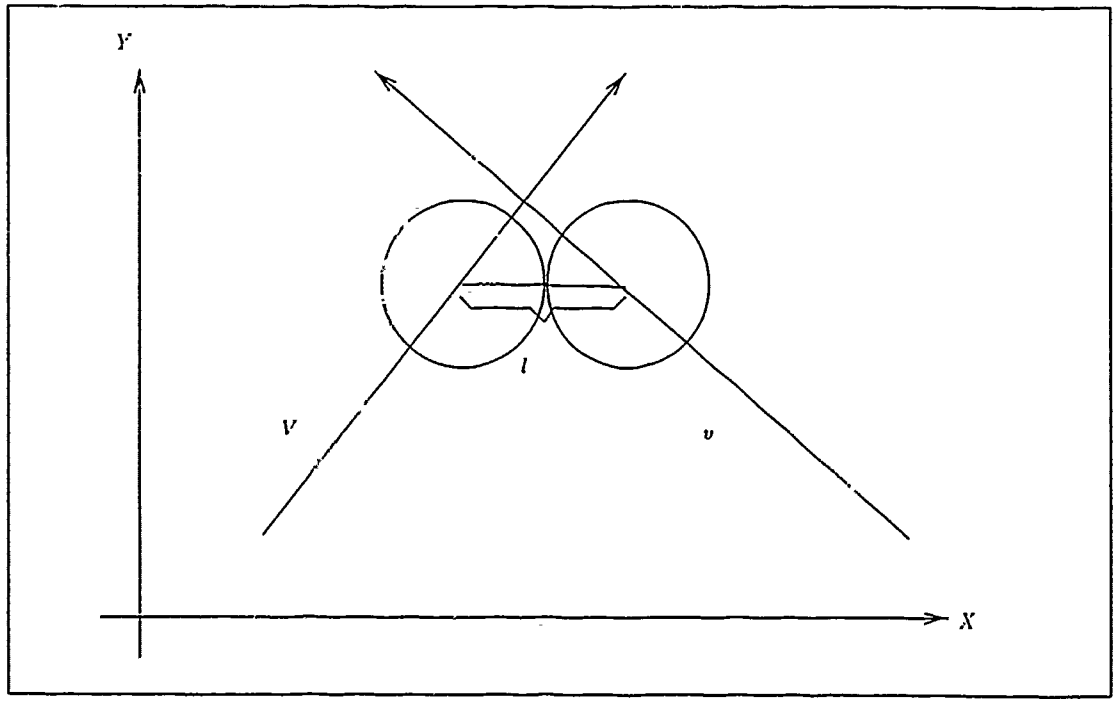


Figure 5. Two Balls Colliding at the Point of Impact

3.3.3.1 Calculating the time of Impact for a Ball Collision The location of a pool ball is defined by the coordinates of its center of mass; therefore, the linear distance separating two balls at the moment of impact is the sum of the radii. Since every pool ball for this simulation has equal radius, the value of l^2 in equation (8) is $4R^2$. Equation

(8) is most easily implemented if the two colliding pool balls have the same initial logical times; otherwise, the difference in logical times must be accounted for as variables. The possibility exists that two colliding pool balls could have different initial logical times. To accommodate this possibility, the pool ball having the lesser logical time is moved by a delta t such that its logical time is equal to that of the other pool ball. The values of X_1, x_1, Y_1 and y_1 (after both pool balls have equal logical times) must be substituted with those of equation (5). Solving for ΔT results in equation (9).

$$0 = a\Delta T^2 + b\Delta T + c \tag{9}$$

where

$$\begin{aligned} a &= V_x^2 + V_y^2 - 2v_xV_x - 2v_yV_y + v_x^2 + v_y^2 \\ b &= 2X_0V_x + 2Y_0V_y - 2X_0v_x - 2Y_0v_y - \\ &\quad 2x_0V_x - 2y_0V_y + 2x_0v_x + 2y_0v_y \\ c &= X_0^2 + Y_0^2 - 2x_0X_0 - 2y_0Y_0 + x_0^2 + y_0^2 - 4R^2 \end{aligned}$$

The values of a, b and c are simply the coefficients to the quadratic formula from which ΔT may easily be solved. From an algorithmic point of view, only real roots to the quadratic solution represent viable collision times. As such, the determinant must be inspected for non-negative values. If the quadratic solution consists of two real roots, the lesser of the two represents the delta time at which the two pool balls in question will collide.

3.3.3.2 Calculating the State Information after a Ball Collision Once two pool balls collide with one another, both velocity vectors will change. Solving for the new velocity vectors in the X/Y coordinate system is most easily solved if the coordinate system is rotated to form a new R/P orthogonal system where R is the axis formed by the tangent to the two pool balls at the point of collision and P is the axis which is perpendicular to R. It will be shown that the vector components V_R and v_R are simply interchanged as a result of a pool ball collision.

To solve for the new velocity vectors, equations (3) and (4) are used where $|\vec{v}_0|^2 = v_{R0}^2 + v_{P0}^2$ and $|\vec{V}_0|^2 = V_{R0}^2 + V_{P0}^2$ in equation (3). After substituting the values of $|\vec{v}_0|^2$ and $|\vec{V}_0|^2$ into equation (3), the equations for conservation of energy and momentum become

that of equation (10) and (11) in terms of the new R/P coordinate system.

$$\begin{bmatrix} v_{R0}^2 + V_{R0}^2 \\ v_{P0}^2 + V_{P0}^2 \end{bmatrix} = \begin{bmatrix} v_{R1}^2 + V_{R1}^2 \\ v_{P1}^2 + V_{P1}^2 \end{bmatrix} \tag{10}$$

$$\begin{bmatrix} v_{R0} + V_{R0} \\ v_{P0} + V_{P0} \end{bmatrix} = \begin{bmatrix} v_{R1} + V_{R1} \\ v_{P1} + V_{P1} \end{bmatrix} \tag{11}$$

Equations 11 and 10 can be easily manipulated to produce equation 12.

$$\begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} (V_{R1} - v_{R0})(V_{R1} - V_{R0}) \\ (V_{P1} - v_{P0})(V_{P1} - V_{P0}) \end{bmatrix} \tag{12}$$

Solving for V_{R1} and V_{P1} yields $V_{R1} = v_{R0}$ and $V_{P1} = V_{P0}$. Substituting these values into equation (11) results in equation (14) which represents the resultant velocity vectors of the two colliding pool balls in terms of the R/P coordinate system.

$$\begin{bmatrix} v_{R1} \\ v_{P1} \end{bmatrix} = \begin{bmatrix} V_{R0} \\ v_{P0} \end{bmatrix} \tag{13}$$

$$\begin{bmatrix} V_{R1} \\ V_{P1} \end{bmatrix} = \begin{bmatrix} v_{R0} \\ V_{P0} \end{bmatrix} \tag{14}$$

3.3.3.3 Rotating the X/Y Coordinate System Figure 6 illustrates the rotation of the X/Y coordinate system to form the new R/P coordinate system. The line connecting the two pool ball's centers is one of the desired orthogonal axes. Let this line be L. Let θ be the angle made with this line and the X-axis. Then

$$\theta = \arctan\left(\frac{Y_1 - y_1}{X_1 - x_1}\right) \tag{15}$$

Let ϕ be the angle made with the velocity vector V and the X-axis. Then

$$\phi = \arctan\left(\frac{V_y}{V_x}\right) \tag{16}$$

Let φ be the angle made with the velocity vector V and the line L. Then

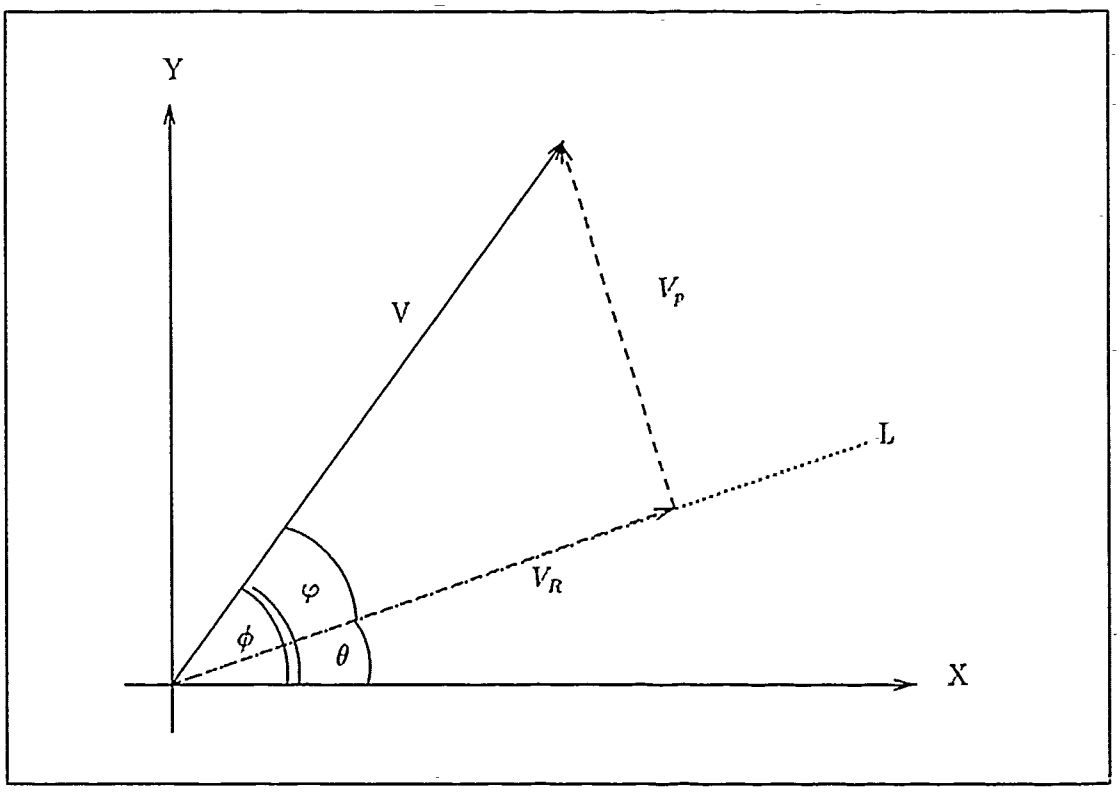


Figure 6. Translating an X/Y Axis System

$$\varphi = \phi - \theta \tag{17}$$

The new orthogonal reference system has axes R and P where R is the axis along the line L connecting the two balls and P is perpendicular to R. The new velocity vector of a given ball is then defined by V_R and V_P which are clearly the following.

$$\begin{bmatrix} V_R \\ V_P \end{bmatrix} = \begin{bmatrix} V\cos(\varphi) \\ v\sin(\varphi) \end{bmatrix} \tag{18}$$

The velocity components V_R and v_R may now be interchanged in accordance with equation (14); however, it is more convenient to first decompose V_R and V_P into their

respective X and Y components yielding equation (19).

$$\begin{bmatrix} V_{xR} \\ V_{yR} \\ V_{xP} \\ V_{yP} \end{bmatrix} = \begin{bmatrix} V_R \cos(\theta) \\ V_R \sin(\theta) \\ -V_R \sin(\theta) \\ V_R \cos(\theta) \end{bmatrix} \tag{19}$$

Once V_{xr} and V_{yr} are interchanged with v_{xr} and v_{yr}, the resultant velocities must be translated back into the X/Y orthogonal axis system resulting in equation (20).

$$\begin{bmatrix} V_x \\ V_y \\ v_x \\ v_y \end{bmatrix} = \begin{bmatrix} v_{xR} + V_{xP} \\ v_{yR} + V_{yP} \\ V_{xR} + v_{xP} \\ V_{yR} + v_{yP} \end{bmatrix} \tag{20}$$

3.4 Simulation Environment

A simulation environment existed for all distributed discrete event applications. This environment was required to be used in an effort to standardize software from various research efforts. The environment is object oriented in the C programming language. The objects defined in the environment include a simulation driver, a clock, a next event queue and a generic event which must be tailored to a specific application.

3.4.1 Simulation Driver

Functional Description The *driver* forms a basic conditional loop construct. At each iteration, an event is executed, a new event determined, and another event is executed until a DONE event is reached. A DONE event signifies that the execution of another event would set the simulation clock beyond the user specified simulation run time. The loop exits, and control of the program is returned to the iPSC/2 host processor.

Functional Description: The *clock* object manages all aspects of the simulation clock. The simulation time is updated each time the clock object is called. The simulation time is a double precision floating point variable.

Attributes: Time.

Operations:

- init_time
- set_time
- adv_time
- get_time

3.4.2 The Next Event Queue Object

Functional Description: The *next event queue* (NEQ) object stores all scheduled events. The events are insertion sorted by simulation time (next event time). The NEQ is implemented as a singularly linked list.

Attributes: None.

Operations:

- init_neq
- show_neq
- add_event
- get_event
- count_event
- neq_time
- simultaneous
- count

3.4.3 The Event Object

Functional Description: The *Event* object as implemented is actually a class of objects such that each desired event must be instantiated from the 'event' operations. Each instantiation becomes an object.

Attributes:

- Time
- ID
- Type of event
- Pool Ball ID(s)

Operations:

- new_event (allocates memory for an event)
- show_event
- zap_event (deletes memory allocation for an event)

IV. Software Design

4.1 Introduction

This chapter outlines the design steps used to develop the parallel pool balls simulation. The pool balls simulation was designed in three incremental steps.

1. Development of a sequential simulation without partitioning or data replication.
2. Development of a sequential simulation with partitioning and data replication.
3. Development of a parallel simulation with partitioning, data replication and conservative synchronization.

Each of the above designs were object oriented. An object oriented design (OOD) approach was selected to enhance software maintenance capabilities for follow-on research of the pool balls concept. This chapter describes the evolution of the sequential, non-partitioned simulation into the parallel implementation by first analyzing the sequential version and then by highlighting the design changes required to implement the follow on versions.

4.2 Design of a Sequential Simulation without Spatial Partitioning

The simulation environment was object oriented. An object oriented application was therefore easier to interface than alternative designs such as functional, top down, or Jackson (JSD). The OOD pool balls application defined objects representing a ball object, ball object manager, table sector, table sector manager, random number generator and an event handler object. The *table* object creates the pool table for the appropriate or specified table dimensions and stores all of the boundary information related to the table. The *ball* object creates all of the pool balls for the simulation and stores them in a data structure. All operations related to the management of pool ball objects take place here. The *random number generator* is a machine independent pseudo-random number generator developed by Law and Kelton (14). This random number generator is capable of generating uniform, logarithmic, exponential and normal distributions having a lower and upper bound. The *event_handler* object determines the next event of interest and enforces an event passed to it.

4.2.1 Design of the Ball Object

Functional Description: The *ball* object is an instantiation of a class of pool balls. Each pool ball object is dynamically allocated and deallocated to and from memory.

Attributes:

- Radius
- Ball_ID
- Ball_Time
- X
- Y
- Vx
- Vy
- Do_I_See_It
- Do_I_Own_It
- Who_Owns_It

Operations: None.

4.2.2 Design of the Ball Object Manager

Functional Description: The *ball manager* is an abstract data type whose sole purpose is to store the pool balls which are assigned to it.

Attributes: None.

Operations:

- Initialize
- Add
- Remove
- Reset
- Increment
- Set
- More

- Get_and_Delete
- Get_Next_Ball
- Get_This_Ball
- Print_Balls_In_This_Sector
- Head
- Tail
- Is_Empty
- Is_Found
- Length_Of
- Check_This_Ball

Detailed Design: It was desirable to design data structures for the sequential, non-partitioned application which required as few changes as possible to accommodate the more complex sequential, partitioned version and the parallel, partitioned version of the software. It was also desired to minimize the search, add and delete functions for the ball object manager. These functions require searching and traversing the data structure which stores the ball objects. For the relatively simple case of the sequential, non-partitioned application, an array structure minimizes the search and traversal time. This is not the case for a partitioned application because each sector has a high probability of containing only a subset of pool balls. Let N be the total number of pool balls created and let M be the number of pool balls in any given sector at some instant in time. Then $M \leq N$. Three data structures were considered during the design phase: an array, a linked list and an indexed linked list. The time complexities for each of the three data structures considered are shown in Table 1 for the search and traversal operations.

Table 1. Order of Analysis for the Ball Manager Data Structure

Data Structures	Time to Search	Time to Traverse
Array	$O(1)$	$O(N)$
Linked List	$O(M)$	$O(M)$
Indexed LL	$O(1)$	$O(M)$

From Table 1, the indexed linked list provides superior time complexity for both searching and traversing, especially when $N \gg M$. Therefore, the indexed linked list data

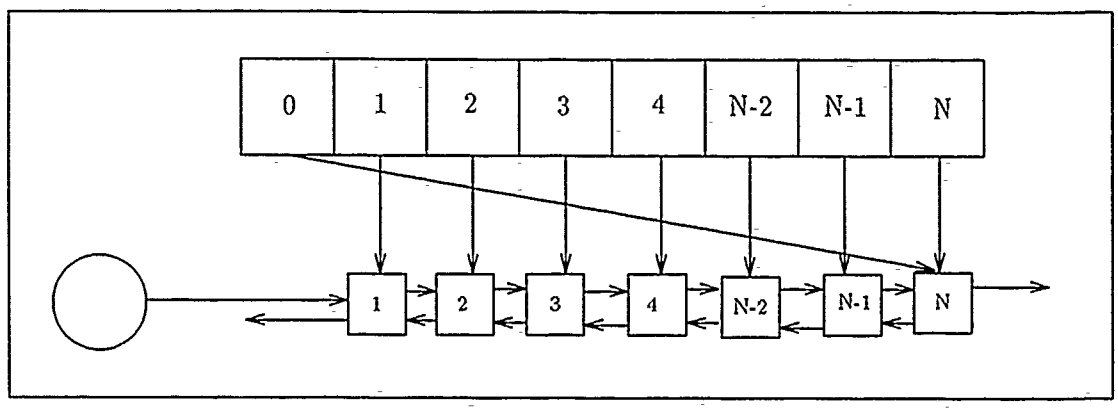


Figure 7. Data Structure for Storing Pool Balls

structure was selected even though the initial application uses only a single sector, negating any advantages of the indexed linked list over the simple array. The data structure is shown pictorially in Figure 7. All ball objects contain a Ball_ID attribute which is implemented with unique positive integers. The array element corresponding to zero is set to always point to the tail of the linked list.

4.2.3 Design of the Table Sector Object

Functional Description: The *table sector* object is an instantiation of a class of sectors. Even though the initial application did not incorporate sectoring, it was desirable to develop data structures which were easily transportable to the more complex applications to be designed later. For the inital software version, the pool table consisted of a single 'sector'. The sector object defines the sector boundaries. Each sector is assigned a unique Sector_ID number.

Attributes:

- Sector_ID
- Left_Border
- Right_Border
- Top
- Bottom
- Type_Left_Border
- Type_Right_Border

- Left_Neighbor
- Right_Neighbor
- Is_Left
- Is_Right

Operations: None.

4.2.4 Design of the Table Sector Manager

Functional Description: The *table sector manager* object stores all of the sector objects and provides information about the sectors.

Attributes:

- Table_Length
- Table_Width

Operations:

- Determine_Table_Dimensions
- Determine_Sectors
- Get_X_Length
- Get_Y_Length
- Get_Left_Border
- Get_Right_Border
- Get_Top
- Get_Bottom
- Get_Left_Type
- Get_Right_Type
- Do_I_Have_a_Left_Neighbor
- Do_I_Have_a_Right_Neighbor
- Get_Left_Neighbor
- Get_Right_Neighbor
- Print_Sector
- Print_All_Sectors

Detailed Design: Each pool ball has a finite radius and the pool table has a finite area; therefore, for any given pool ball radius and table area, there is an upper bound on the number of pool balls which can fit on the table without overlap. Three methods were considered during the design phase to determine the length and width of the table.

1. command line argument
2. constant
3. dynamic calculation

The first option was deemed too awkward to work with from a user's point of view. The second option was favored because it is easy to implement and requires no computations; however, recompiling becomes necessary if the table dimensions are too small to accommodate a desired quantity of pool ball objects. The last option avoids recompiling. Both options two and three above were finally selected by adding a single command line argument. The default was set to a constant value for length and width. The variable dimension option (if selected) dynamically calculates the length and width of the table by using a constant density formula. The density of pool balls to table area was defined by taking the ratio of a the area occupied by 16 pool balls, each having a one-inch radius, to the area of a 6 foot by 12 foot table. The table length was defined to be twice the table width; therefore, the known table density and the user specified quantity of pool balls dynamically determines the length and width of the pool table if the variable table dimension option is selected in the command line arguments.

4.2.5 Design of the Random Number Generator Object

Functional Description: The *random number generator* object was borrowed from the works of Law and Kelton (14). This object is a machine independent pseudo-random number generator which produces a stream of random numbers given an input seed.

Attributes: Seed.

Operations:

- Uniform
- Exponential
- Normal
- Lognormal

4.2.6 Design of the EventHandler Object

Functional Description: The *EventHandler* object determines the next event for a given sector and executes an event which is passed to it.

Attributes: None.

Operations:

- Initialize
- Determine_Next_Event
- Execute_Next_Event

4.2.7 Event Definitions Four event types are possible for the sequential, non-partitioned version of the pool balls simulation:

1. A pool ball striking a vertical cushion.
2. A pool ball striking a horizontal cushion.
3. A pool ball striking another pool ball.
4. A 'DONE' event indicating that the simulation is over.

The 'CUSHION' events were identified as either horizontal or vertical since the behavior of the collision differs between them. A 'COLLISION' event consists of a collision between two pool balls and the identification of both ball objects is stored with the event information. A 'DONE' event signifies that the next event has an event time greater than the user specified simulation time; therefore, the execution of a DONE event terminates the simulation.

4.2.8 Algorithm Design Two high level algorithms were considered for the design of the pool balls simulation. One has a time and space complexity of $O(N)$ and $O(N)$ while the other has $O(N^2)$ and $O(1)$. AFIT's iPSC/2 hypercube has 4 megabytes of memory per node; therefore, memory space was not considered to be a limiting factor for reasonable quantities of pool balls. The three factors that were considered are granularity, research time and event list manipulation. The iPSC/2 is a course grain machine. DeCegama shows how performance on distributed processors is affected by the granularity of the software

with respect to the machine's grain size (9). He concludes that a fine grain algorithm *must* be implemented on a fine grain machine to achieve reasonable speedup. The pool balls simulation is inherently fine grain. This mis-match in granularity was anticipated to result in poor speedup results. This raises the following question: if speedup results are poor, does this conclude that spatially partitionable models should not be implemented with conservative paradigms, or does it simply reinforce the assertion that fine grain algorithms should not be implemented on course grain machines? The latter statement fails to address any of the objectives of this thesis. The following options were available:

1. Use the $O(N)$ algorithm on the iPSC/2 knowing that the mis-match in granularity exists.
2. Use the $O(N)$ algorithm on a fine grain machine.
3. Use the $O(N)$ algorithm on the iPSC/2, but incorporate spin loops to artificially raise the computational complexity of the algorithm. This increases the computations/communications ratio which changes the granularity of the algorithm from fine to course.
4. Use the the $O(N^2)$ algorithm on the iPSC/2 which also artificially raises the computations/communications ratio.

The first option was dismissed because it fails to address the objectives of this thesis. The second option was not possible because AFIT has only course grain machines (iPSC/1 and iPSC/2). The third and fourth options are both viable and both were analyzed carefully.

The $O(N)$ algorithm was considered to be more difficult to implement and therefore would require more time to design, implement and debug. The $O(N)$ algorithm requires efficient storage of future events known as event list manipulation. Misra has shown that event list manipulation is the limiting factor toward speedup (18). The $O(N^2)$ algorithm avoids complex event list manipulation by storing only one event at a time. This is relatively easy to implement. In the final analysis, time was considered the limiting factor; therefore, option four was selected.

The basic algorithm for the pool balls simulation revolves around a loop construct. In each loop, every ball is analyzed to determine which ball will have the minimum next event time. This event is scheduled by inserting it into the next event queue; thus, the next event queue is refreshed in every loop. Simultaneous events are both possible and allowed in which case multiple events are inserted into the next event queue. After scheduling the

next event, the event is removed from the queue and executed. Execution of an event consists of removing the appropriate pool balls(s) from the ball object manager, updating the new position, calculating a new velocity, updating the pool ball's time stamp and returning the pool ball to the ball object manager. The next event queue is checked for additional (simultaneous) events. After the next event queue is confirmed to be empty, the loop starts over again. The simulation ends if the next event stored in the next event queue is a DONE event. A DONE event is inserted into the next event queue if, in the determination of the minimum next event, the next event time corresponding to the next event is greater than the user specified simulation time. This time is specified as a command line argument. The algorithm may be summarized by the following loop.

```
While (! Done) loop
    1. Determine the next event(s).
    2. Schedule the next event(s).
    3. while (! empty) loop
        if (Type ≠ DONE)
            Execute the next event
        else
            Done = TRUE
End loop
```

The basic algorithm used in this simulation has a time complexity of $O(N^2)$ and a space complexity of $O(1)$. The time complexity stems from the fact that each pool ball must inspect every other pool ball on the table to determine if and when a collision will occur. Thus, the first ball must inspect $N-1$ balls, the second ball must inspect $N-2$ pool balls and the $N-1^{th}$ ball must inspect 1 ball. Thus, the time complexity is:

$$\sum_{i=1}^{N} i = \frac{N^2+N}{2}$$

$$\therefore \text{Time Complexity} = O(N^2)$$

The $O(N)$ algorithm, although not implemented, works as follows. During the first loop iteration, each ball will have inspected every other ball. At most N events will be pheasible, one for each ball and each with different event times. These events will be stored in memory. After executing an event involving ball B, all events in the event list containing ball B must be removed. Ball B must then inspect all remaining $(N-1)$ balls

to regenerate the previously removed events. If the event list data structure is designed efficiently, at most $O(N)$ time is required to remove events involving ball B and $O(N)$ time is required to regenerate the new events involving ball B. Thus, the overall time complexity is $O(N)$ instead of $O(N^2)$

4.2.9 Design of the Queue Structures This thesis defines the scheduling of an event to be the insertion of an event into the next event queue. No other events are inserted into the next event queue unless they occur at the same simulation time. As each pool ball is inspected during the determination of the next event, the current minimum next event is temporarily stored. A simple event structure does not suffice because the possibility exists of finding an event whose next event time is *equal* to the current minimum. Such an event cannot be discarded nor can it replace the current minimum next event. Therefore, a *candidate* queue was designed to store the current next event. After all of the pool balls have been inspected for next events, the candidate queue guarantees to contain the absolute minimum next event. This event is then 'scheduled' by placing it on the next event queue.

4.2.10 Version 1 Structure Chart The structure chart for the non-partitioned sequential simulation is shown as Figure 8.

4.2.11 Command Line Arguments The pool balls simulation design incorporates ten command line arguments all of which are optional to the user. Each argument has a default value in the event that an option is not selected. The arguments available are listed in Table 2.

Write to Disk: This option writes the ball state information to disk after each pool ball changes state. This output file allows the user to inspect the data and to compare data runs. This option degrades the run time performance of the simulation not only due to slow disk I/O but also because the host processor performs all of the writing to disk. As each pool ball is moved, the node processor sends the data to the iPSC/2 host processor for writing. Thus, a heavy penalty is extracted for this command line argument.

Error Checking: This option examines each pool ball prior to being moved and was intended for debugging purposes only. A pool ball must lie within the borders of the sector to which it is assigned and the time tag must be both positive and less

Table 2. Command Line Arguments

Argument	Function	Default
-w	write to disk	FALSE
-e	error checking	FALSE
-cp	Continuously print to screen	FALSE
-ip	initially print to screen	FALSE
-b#	number of balls	25
-p#	number of partitions	1
-f	fixed table dimensions	TRUE
-t#	simulation time	60.0 sec
-s	collect statistics	FALSE
-ne#	number of events	50

than the event time associated with the requested event. If any of these items are incorrect, the event is not executed, an error message prints to the screen and the simulation abruptly terminates.

Continuously Print to Screen: This option prints all of the pool ball state information to the screen each time a pool ball is moved. This allows real-time examination of the simulation; however, a hefty penalty is placed on run time performance.

Initially Print to Screen: This option prints the initial state information for all of the pool balls after initialization. There is no penalty in run time performance because the real-time clock is not started until after initialization.

Number of Balls: This option specifies the number of pool balls desired for the simulation.

Number of Partitions: This option specifies the number of partitions desired for the simulation.

Fixed Table Dimensions: This option determines whether the table length and width are set by a pre-defined constant in the software or if a dynamically calculated table length and width are to be used based upon a pre-defined density constant and the number of pool balls selected.

Simulation Time: This option specifies the simulation time.

Collect Statistics: This option writes the simulation run time as a function of the number of pool balls to a file. This file can then be used to plot the results of many test runs without manual data entry.

Number of Events: This option allows an alternate technique to be used to terminate the simulation. Each node of the iPSC/2 keeps track of the number of events that have been processed. When using a single node, the simulation may be set to terminate after processing a specified number of events. This option is not valid when using multiple nodes for the parallel version.

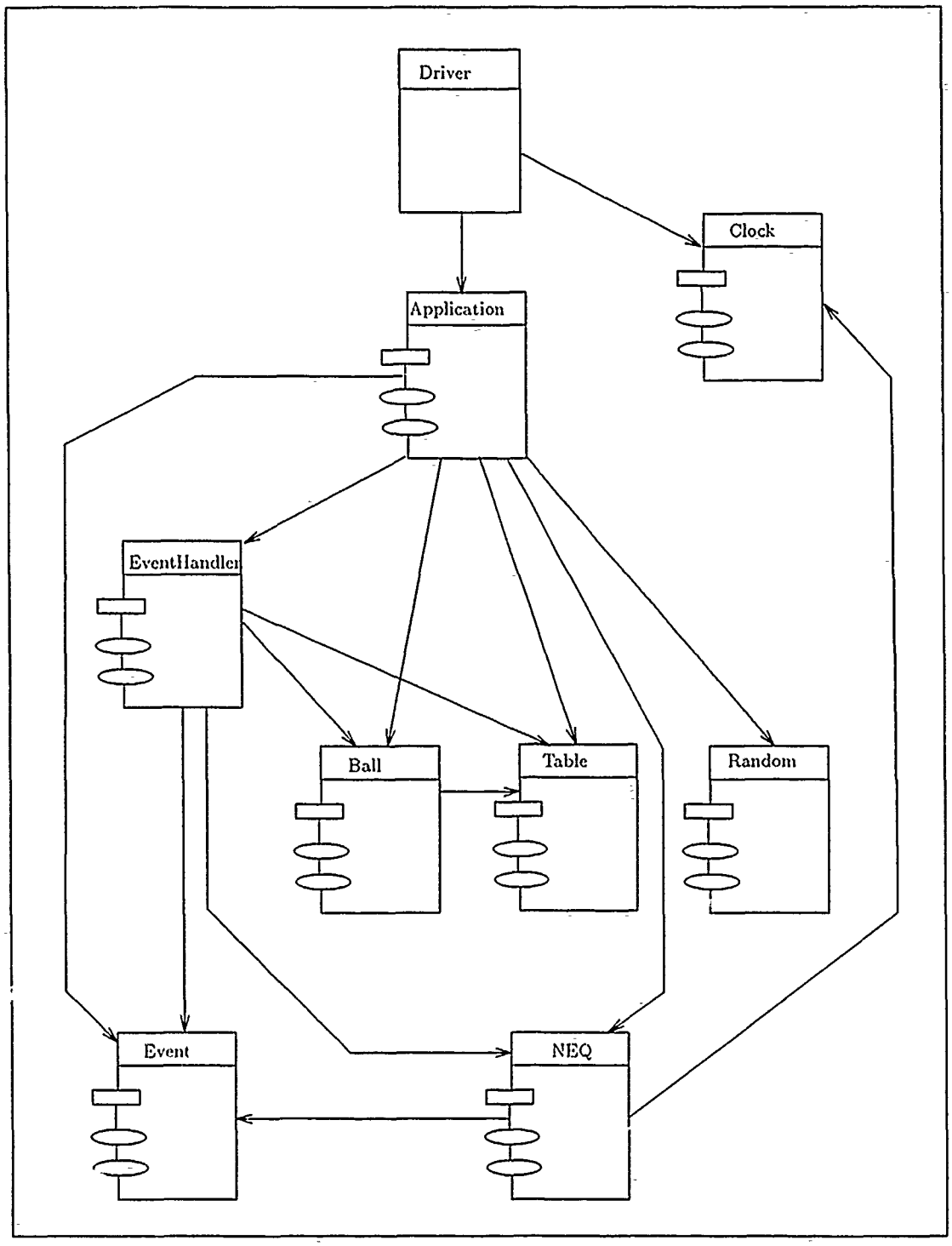


Figure 8. Version 1 Structure Chart

4.3 Design of a Spatially Partitioned Sequential Simulation

The basic design of the second simulation version was the same as the first version; however, additionally functionality had to be incorporated and some of the data structures had to be modified. The high level algorithm was unchanged.

4.3.1 Changes to the Ball Object The previous simulation design incorporated a single indexed linked list for the ball object mangager data structure. The partitioned version requires an indexed linked list *per sector*. To accommodate this change, each indexed linked list is encapsulated into a record structure; thus, each sector has one record type. An array of sectors of length P was designed where P equals the number of sectors. Each array element contained a pointer to the appropriate record type containing the indexed linked list.

4.3.2 Changes to the Table Object The previous sequential simulation called for a simple record structure storing the table's boundary information; however, there was only one sector to contend with. The design of the spatially partitioned version of the sequential simulation incorporates an array of sectors of size P where P equals the number of sectors. Each array element points to a record structure containing individual sector boundaries. The sectors were designed to be equal in length and width. The table is partitioned vertically along the X axis. The table may be partitioned into $P < P_{max}$ where P_{max} is the number of sectors corresponding to a sector width greater than the predefined pool ball diameter.

4.3.3 Changes to Event Definitions In the previous non-partitioned version, it was possible for a pool ball to traverse the entire pool table along the X-axis in one step. The partitioned version requires two or more incremental steps depending upon the number of sectors requested. A pool ball must now stop at a sector border whereupon data replication may take place, followed by a crossing from one sector to the next sector. The former event is defined to be a 'PARTITION' event while the latter event is defined to be an 'EXIT' event. Control of a pool ball normally takes place during an EXIT event. As the number of partitions increase, the number of incremental events increases for pool balls moving in the X direction.

4.3.4 Changes to the Simulation Algorithm The algorithm was changed in the following manner:

1. 'Partition' and 'exit' events were added.
2. Determination of the next event was modified.
3. Enforcement of Wieland's data replication was added.
4. Design of the ball object data structure was modified to allow each sector to have its own ball manager.
5. Design of the table data structure was modified to allow storage of individual sector information.

The basic algorithm now consists of the following steps:

1. Determine the next event.
2. Schedule the next event.
3. Execute the next event.
4. Enforce Data Replication (as needed).

To determine the next event, each sector is inspected one at a time. There is one global candidate next event queue and one global next event queue. As each sector is inspected, the individual events are compared against the global candidate next event queue. The time to determine the next event requires searching P sectors, each containing on average $\frac{N}{P}$ pool balls; thus, the time complexity is reduced from $O(N^2)$ to $O\left(\frac{N^2}{P}\right)$ where N is the total number of pool balls and P is the number of sectors.

Scheduling the next event remains unchanged. After completing the next event determination phase, the single candidate list guarantees to contain the minimum next event for all sectors (the minimum next event for the entire pool table). Scheduling that event consists of removing the candidate event from the candidate queue and inserting it into the next event queue.

Executing the event requires knowledge of the sector from which the next event originated. All of the procedures in the simulation were thus changed to allow this information to be passed as in-type parameters. The sector identified as enforcing the collision retrieves the ball(s) from its ball object manager, updates the position(s), calculates and stores the new velocity trajectories, updates the ball time tag(s) and returns the pool ball(s) to the sector's ball object manager.

This simulation used a two step data replication strategy proposed by Wieland (22). The replication rules were established in a *communicator* object. Ball objects must be replicated or de-replicated under the following conditions:

1. Providing Visibility: A sector must be given visibility of a pool ball if the ball was previously owned by another sector but upon moving, the ball now lies within the border region. The center of the ball must still lie within the adjacent sector; otherwise, the gaining sector not only has visibility of the ball but also has control of the ball.
2. Providing Ownership (Control): A sector must be given control of a pool ball if the ball was previously owned by another sector but upon moving, the ball now has its center of mass within the gaining sector's boundaries.
3. Removing Visibility: A sector must remove a ball from its ball manager object if the ball was previously visible but upon moving, the ball's center of mass lies outside of the losing sector's border region. This implies that the ball is owned by another sector after moving.
4. Removing Ownership (Control): A sector must relinquish control of a pool ball if the ball was previously owned by the losing sector but upon moving, the ball's center of mass now lies beyond the losing sector's boundaries. A ball meeting this condition cannot move any further in one step than the edge of the losing sector's border region. In this manner, control of the ball is passed to the gaining sector but the losing sector retains visibility.
5. Updating Visibility: A sector must have an updated copy of a pool ball if the ball was previously visibile (but not owned) by an adjacent sector and upon moving, the ball is still visible (and still not owned) by the adjacent sector.

Each pool ball object has associated with it a visibility flag and an ownership flag. All pool balls owned by a sector must also be visible to the sector or an error condition is raised. The act of providing visibility of a pool ball to an adjacent sector consists of providing a copy of the pool ball object to the adjacent sector's ball object manager. The only difference between the two copies is the status of the ownership flag. Removing visibility consists of requesting an adjacent sector's ball object manager to delete the replicated ball. Changing of ownership status is analogous. Since this version of the simulation is still implemented on a single processor, all commands may be implemented directly through memory. Messages are not required.

4.4 Design of a Parallel Simulation

The basic design of the parallel simulation was the same as the sequential simulation with spatial partitioning and data replication. Some additional functionality had to be added to enforce the conservative synchronization paradigm and some of the data structures had to be changed to accommodate the distributed environment.

4.4.1 Changes to the Ball Object The partitionable sequential design encapsulated a ball object manager for each sector. An array of length P contained pointers to each sector's ball manager. For the distributed simulation, no one node will ever contain all of the table sectors; therefore, the data structure was changed as follows. Each node has one data structure consisting of an array of pointers representing a list of ball managers. The length of each node's array is $\frac{P}{M}$ where P is the number of table sectors and M is the number of nodes. This ratio represents the number of partitions per node which is defined to be the same for all nodes. Therefore, a design constraint limits the number of sectors to be an even multiple of the number of nodes.

4.4.2 Changes to the Table Object The partitionable sequential design encapsulated each sector's boundary information in an array of length P. The distributed design assigns one array of sector information to each node and the length of each array is reduced from P to $\frac{P}{M}$.

4.4.3 Changes to the Candidate Queue Structure Both sequential simulation designs encapsulated a single candidate next event queue for the entire pool table. The distributed design encapsulates a candidate next event queue for each sector. Each node is allocated an array of candidate queues representing the hierarchical class of queues. The length of each array is $\frac{P}{M}$. This design decision is important because a candidate event is no longer scheduled based solely upon the criterion that it has the smallest next event time. For the distributed simulation, it is highly desirable to schedule as many sectors as possible to achieve efficient parallelism. Obviously if more than one sector is scheduled to execute a candidate event, one event has the smallest next event time while all other scheduled events have a greater next event time. The determination of scheduling candidate events must now reside with determining a minimum safe time per sector. This is discussed later.

4.4.4 Changes to the Next Event Queue Structure Both sequential simulation designs allocated a single next event queue defined by the Institute. The distributed design

must allocate one next event queue per node to reduce internode communications. This is discussed further in Chapter V.

4.4.5 Changes to the Simulation Algorithm The basic high level algorithm was changed to incorporate the minimum safe time calculation. The formulation of a minimum safe time (MST) is explained in detail in Chapter V. The high level algorithm for the parallel application consists of the following steps:

While (! Done) loop
1. For each node in parallel, determine the candidate next event.
2. For each node in lockstep, determine the minimum safe time.
3. For each node in parallel, schedule a candidate event if and only if a sector's candidate next event time is less than or equal to the sector's calculated minimum safe time.
4. While (! Empty) loop
   If (Type $\neq$ DONE)
      For each node in parallel, execute the next event.
   else
      Done = TRUE
5. For each node in lockstep, enforce Wieland's data replication strategy.

End Loop

The above algorithm is repeated in a loop until the next event time is greater than the user specified simulation time. The loop thus created is performed in lock step synchronizing at exactly two points. Steps 2 and 5 must be in lock step and are performed sequentially. Parallelism may be achieved during steps 1,3 and 5. For the case of step 1, not only is parallelism achieved, but the search space per sector is reduced from $O(N^2)$ to $O(\frac{N^2}{P})$. The gains produced from the decreased search time and parallelism are reduced by the increased number of incremental events created by PARTITION and EXIT event types. These events are required for simulation correctness but are not of real interest in the simulation. Each additional sector adds two more incremental steps for a pool ball to traverse the table in the X-axis direction. This increase is not linear because a multi-partitioned table can result in more than one candidate event meeting its minimum safe time. Thus, one search can yield multiple event executions.

The software design of the parallel pool balls simulation is represented by the leveled data flow diagrams of Figures 9, 10 and 11. The process bubbles having multiple, overlayed

bubbles represent processes that execute in parallel. There is no current standard for parallel data flow diagrams.

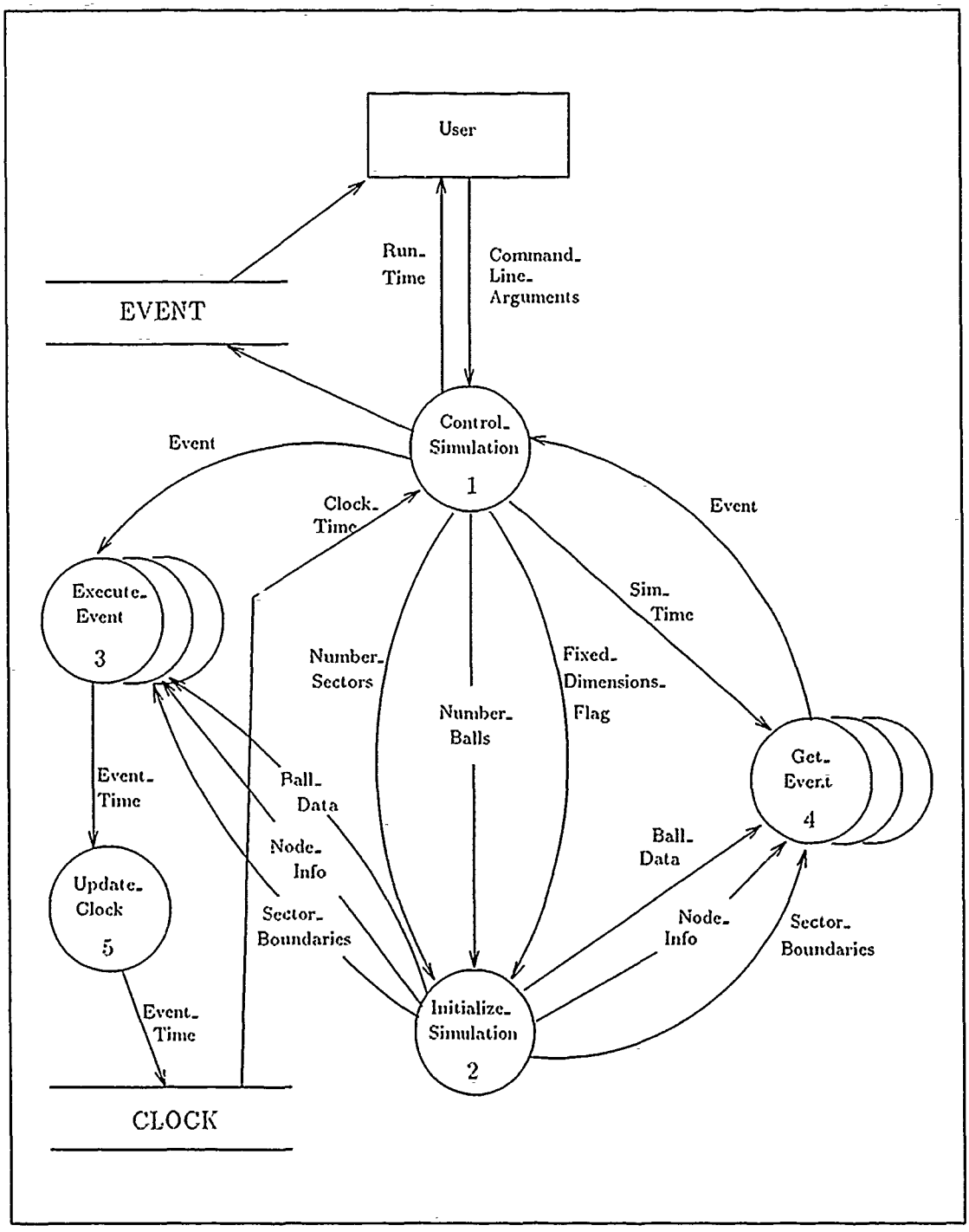


Figure 9. Level 1 Data Flow Diagram

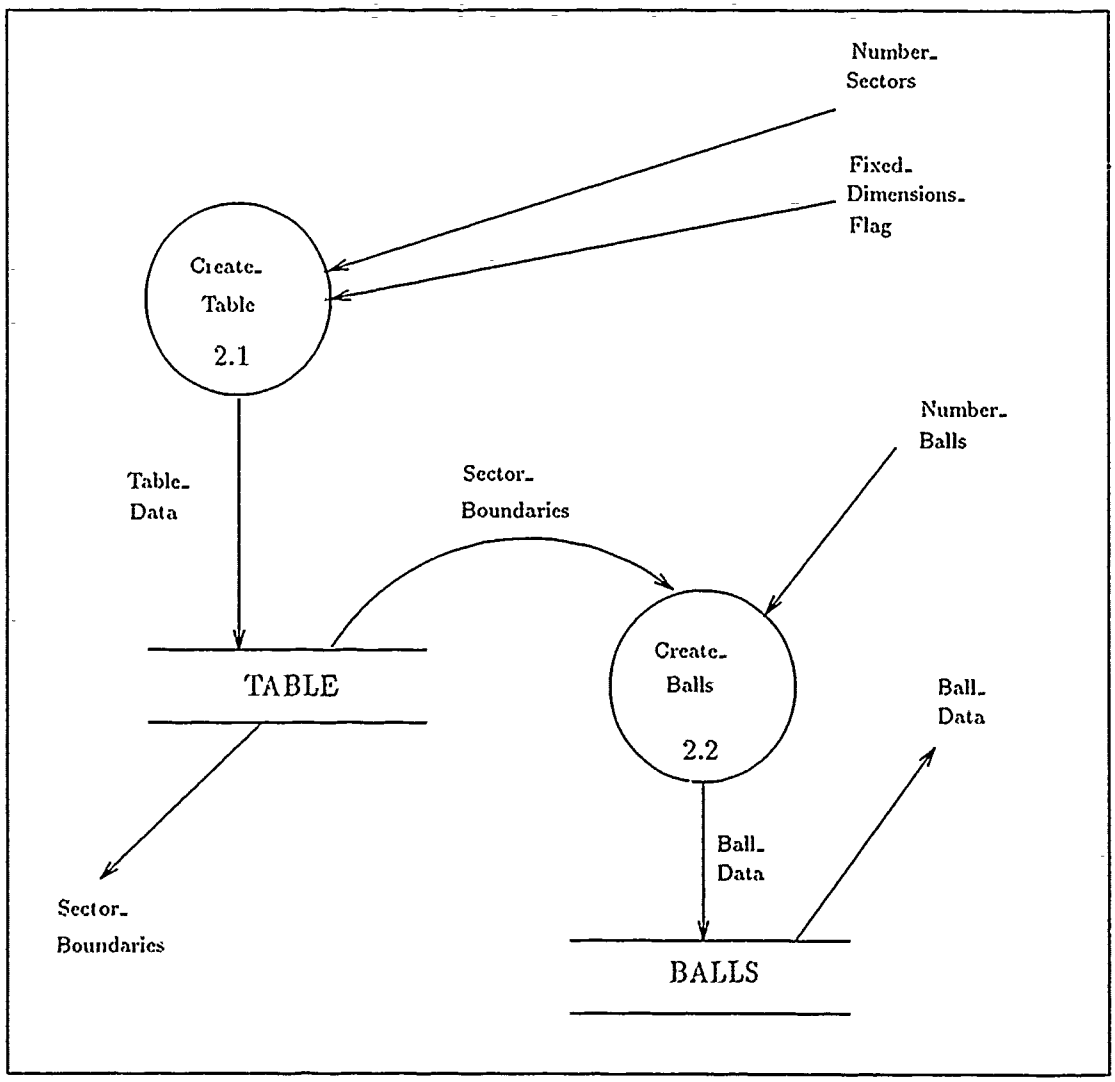


Figure 10. Level 2 Data Flow Diagram for Process 2.0

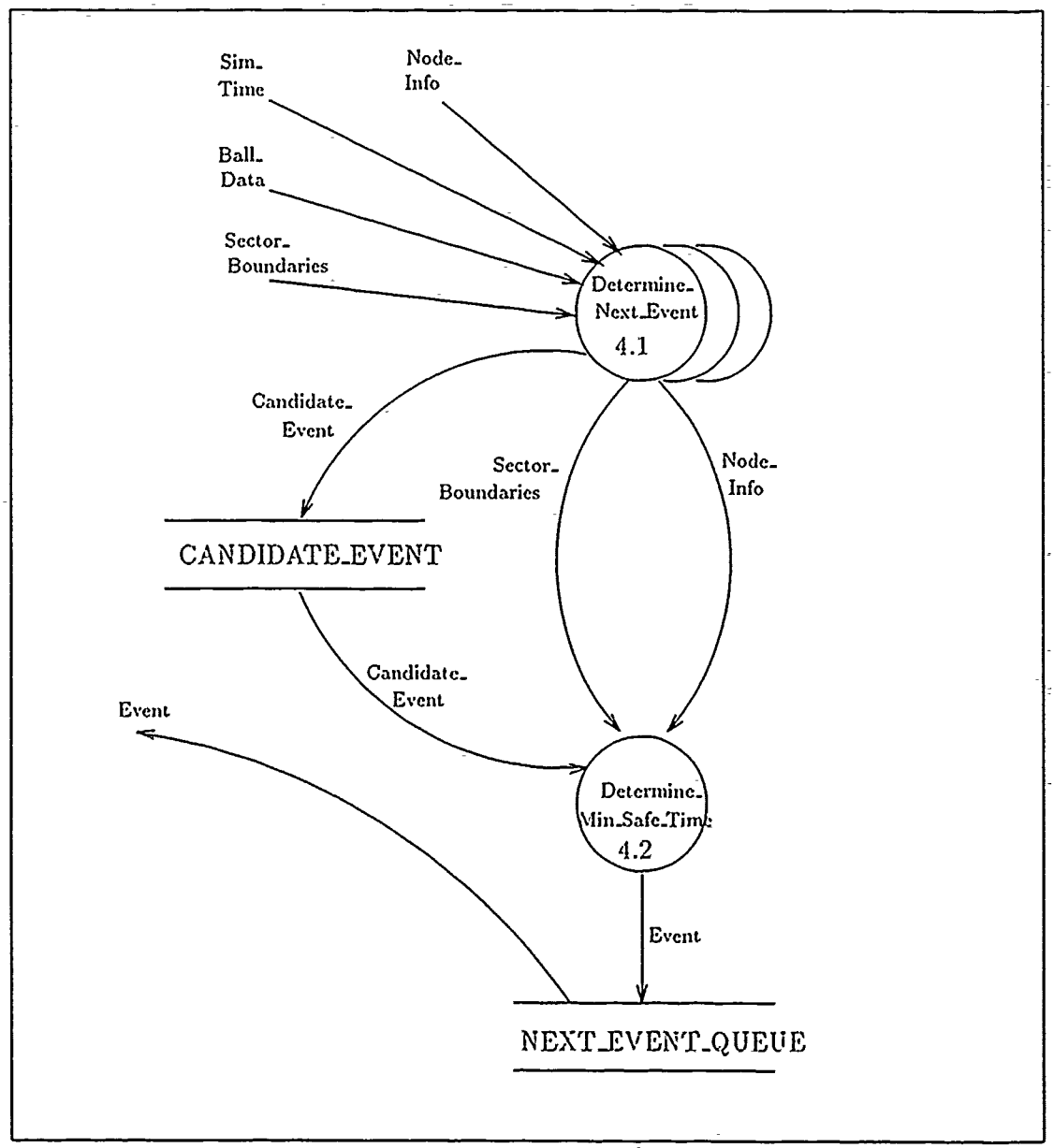


Figure 11. Level 2 Data Flow Diagram for Process 4.0

V. *Parallel Simulation Design and Implementation*

5.1 Introduction

This chapter discusses the design of the pool balls simulation model and the algorithm used to enforce distributed simulation synchronization in a conservative environment. Alternative algorithms are discussed and their respective advantages and disadvantages are analyzed.

5.2 Design of a Parallel Simulation Model

Design of a distributed discrete event simulation requires selection of an appropriate model. Schruben's concepts of *transient* and *resident* models were analyzed for consideration (20). The following analysis explains why the resident entity model was selected.

5.2.1 Modeling Pool Balls as Transient Entities With this model, each pool ball is an entity and parallelism is possible by distributing the pool balls to varying nodes of a distributed processor. In order for a node to determine if a collision will occur between one of its ball objects and another ball object, the node must know of the existence of the other ball. This could be done in any of several manners.

A simple approach is to appoint a central manager. This manager has the state information concerning every ball. Nodes that need to gain access to ball state information request the information from the central manager. If there are M nodes and N pool balls, this approach would require each node to cyclically communicate N messages. With M nodes, the time complexity for communications is $O(N \log P)$. This approach requires each node to wait upon the central manager thus forming a bottleneck. The run time performance is then reduced to the speed of the central manager, thereby driving the simulation to approach sequential performance (3). In terms of search time, each node would require a complete search of all N pool balls. The best possible algorithm incorporating all N pool balls is $O(N)$. The algorithm used in this thesis would require $O(N^2)$ time.

An alternative approach to modeling the pool balls as transient entities assigns a unique subset of N pool balls to each node such that $S_i \cap S_j = \emptyset$, $S_i \cup S_j = S$ and $S_i, S_j \subset S$ where S is the set of N pool balls. Each node is provided a mapping of pool balls to nodes. A table look-up function provides each node with the capability of requesting ball state information directly from the appropriate owner. If the N pool balls

are evenly divided between M nodes, then each node must cyclically request ball state information from all other nodes. Therefore, each node communicates *M-1* times to the remaining *M-1* nodes and each node must provide $\frac{N}{M}$ pool balls. If every node requires the same quantity of messages, the time complexity for this model is $O(N \log M)$. While the time complexity appears to be the same between this approach and the central manager approach, this technique is actually superior because there are no bottlenecks to form. Each node would still have $O(N)$ search space for which to determine a candidate next event.

A third approach assigns a copy of all N pool balls to each node. Each node maintains two lists such that S_i is a set of pool balls owned by $node_i$ and S_j is a set of pool balls not owned by $node_i$ where $S_i, S_j \subset N$ and $(S_i \cap S_j = \emptyset) \wedge (S_i \cup S_j = S)$, S being the set of all N pool balls. As each node changes the ball state information of a pool ball in the set S_i, it must broadcast the state change to all other nodes maintaining a copy of $ball_i \in S_j$. Given M nodes, each event execution requires M messages communicated in $O(\log M)$ time. If every node can process simultaneously, then the communications time approaches $O(M \log M)$. Since every node maintains the set S of all pool balls, the search time is at best $O(N)$. Bottlenecks do not occur. This approach was considered for the Sharks World simulation but rejected during the design phase (8).

5.2.2 Modeling the Pool Table as Multiple Resident Entities With this model, the pool table is partitioned into multiple slices. The slices can be of any shape and can be one or two dimensional. The following discussion considers only one dimensional partitioning, such as slicing the pool table along the X or Y axis.

If the pool balls are uniformly distributed, each node will have approximately $\frac{N}{M}$ pool balls. If $\forall\, i$, $node_i$ has every pool ball lying within a border region, then using Wieland's data replication strategy $\frac{N}{M}$ pool balls must be replicated between adjacent nodes via message passing. Given M nodes, the communications time complexity worst case is $O(N)$; however, this depends upon the predicate that each $node_i$ has all $\frac{N}{M}$ pool balls lying within border regions. Furthermore, each node need only communicate $\frac{N}{M}$ pool balls once during initialization. After initialization, at most two pool balls per sector can move (representing a collision event). If each sector can execute an event simultaneously, then at most $O(M)$ messages must be sent. This is based upon the constraint that nodes need only communicate with their nearest neighbor. This time complexity is further reduced based upon the probability that a sector will have all of its $\frac{N}{M}$ pool balls lying within a

border region. Therefore, the communications time is better than $O(M)$. If each node is partitioned into multiple sectors then each sector has approximately $\frac{N}{P}$ pool balls where P is the number of sectors for the entire pool table. If each *sector*, has every pool ball lying within a border region, and if sectors assigned to the same node are adjacent to one another and communicate via memory at memory speeds, then the initial communications time reduces to $O(\frac{N}{P} \cdot M)$. Since $P > M$, the communications time is less than $O(N)$ with or without parallelism. If $P \gg M$, the communications approaches constant time. Adding potential parallelism in communications and the probability of having nominal percentages of pool balls lying within border regions results in near constant communications time even for $P > M$. In terms of search time, the time complexity reduces from $O(N)$ best case to $O(\frac{N}{P})$, $P \geq M$ and reduces to $O(\frac{N^2}{P})$ using the $O(N^2)$ algorithm presented in Chapter IV. Therefore, the resident entity model is superior to the transient entity model.

5.3 Developing the Minimum Safe Time Calculation

A conservative synchronization paradigm requires a logical process to postpone the execution of an event if there is a possibility of receiving an out-of-sequence message. Each out-of-sequence message has an associated time stamp. This chapter will show that it is not possible to exactly calculate the value of the time stamp for an out-of-sequence message; however, it is possible to estimate it. An estimator is shown to be valid if it guarantees to be less than or equal to the time of arrival of the first transient ball message. Bounds are placed on the estimate from which an estimator is proven to be valid. Simulation progress using the estimator is also proved. This chapter develops and presents three unique estimators all of which are valid; however, only two of them guarantee progress. These two estimators are analyzed and compared. Each estimator has advantages and disadvantages. This chapter will state which of the estimators was selected for design implementation and why.

Definition 5.1: A *transient ball message* is a message containing a pool ball and all of its assiciated state information sent from one sector to another as a result of the pool ball crossing a sector border. Let u be the time of a transient ball message, ν be a discrete time interval representing the simulation iteration number, and i be the sector which receives it; then, $u_i(\nu)$ is the time of arrival at sector i of the next chronological transient ball message. If the simulation time of sector i is w_i, then the condition $u_i(\nu) < w_i$ defines the transient ball message corresponding to $u_i(\nu)$ to be an out-of-sequence message.

Definition 5.2: The *set of event types* for the pool balls simulation consists of five types; partition, exit, collision, vertical cushion, and horizontal cushion events. Let the set of event types be denoted by S; then

$$S \triangleq \{PART, EXIT, COLL, VERT, HOR\} \tag{21}$$

Definition 5.3: An event E in sector i is defined as the tuple

$$E_i \triangleq (t_i(e_i, \nu), B) \tag{22}$$

where

- e_i is an event type $e_i \in S$
- ν is a discrete time denoting the simulation iteration number,
- $t_i(e_i, \nu)$ is the time of the event E_i, and
- B is the set of pool balls associated with the event E_i

Definition 5.4: The *Minimum Safe Time* for sector i is an estimate of the time of the next transient ball message to be received by i. This estimate is denoted $MST_i(\nu)$. The three estimates developed by this thesis are denoted ($MST_i^1(\nu)$, $MST_i^2(\nu)$, and $MST_i^3(\nu)$).

Lemma 5.1: The minimum safe time in sector i must be greater than zero and less than or equal to the time of the first transient ball message received by sector i to be a valid estimate of $u_i(\nu)$. This is stated mathematically as

$$0 < MST_i(\nu) \leq u_i(\nu) \tag{23}$$

Proof: Due to monotonicity of events, the time of arrival of the first transient ball message must be greater than zero. By definition of conservative, a logical process can only execute an event if the event time is less than or equal to the time of arrival of the next transient ball message. Suppose that the next event time $t_i(e_i, \nu)$ were greater than $u_i(\nu)$; then, execution

of E_i would not be allowed. Suppose also that an estimate of $u_i(\nu)$ were calculated. Let this estimate be $MST_i(\nu)$ such that $MST_i(\nu) > u_i(\nu)$. If the condition existed such that $t_i(e_i, \nu) \leq MST_i(\nu)$ resulting in the execution of E_i, then the conservative property would be violated and simulation correctness no longer guaranteed. Therefore, an estimator of $u_i(\nu)$ must be less than or equal to $u_i(\nu)$.

Definition 5.5: P is defined as the predicate of executing an event $E_i = (t_i(e_i, \nu), B)$ such that

$$P(E_i) \triangleq \begin{cases} TRUE & \text{if } t_i(e_i, \nu) \leq MST_i(\nu) \\ FALSE & \text{otherwise} \end{cases} \tag{24}$$

Definition 5.6: At any instant in time, sector i has a set of pool balls in i. Denote this set of pool balls $G_i(\nu)$; then

$$G_i(\nu) \triangleq \{b_1, b_2, b_3, \cdots\} \tag{25}$$

where b_j is a pool ball with ball identification number j. All values of j are unique.

Definition 5.7: By design, every sector in the simulation has the same dimensions. Every pool ball has a minimum *time to cross* a sector based upon its X-axis velocity vector. Then

$$TTC_i(e_i, \nu) \triangleq \frac{W}{V_x(B)} \tag{26}$$

where e_i is an event type in S, ν is an iteration number, W is the width of a sector, B is a pool ball in G_i corresponding to the event $E_i = (t_i(e_i, \nu), B)$, and $V_x(B)$ is the X-axis velocity of the pool ball in E_i.

Lemma 5.2: The time of arrival at sector i of the first transient ball message, $u_i(\nu)$, is greater than or equal to

$$u_i(\nu) \geq \min\{t_{i\pm n}(e_{i\pm n} = PART, \nu) + (n-1) * TTC_{i\pm n}(e_{i\pm n} = PART, \nu)\} \tag{27}$$
$$\forall n : n \neq 0,\ i - n \geq 0,\ i + n < P$$

where $i \in \{0, 1, 2, \ldots, (P-1)\}$ and P is the number of sectors specified by the user.

Proof: Given the set of event types S, a ball cannot arbitrarily enter sector i from sector j without first being scheduled for a partition event in j. This premise is based on the definition of Wieland's two step sectoring strategy. A pool ball in a sector adjacent to sector i produces a transient ball message to sector i as a direct consequence of a partition event $E_{(i\pm1)} = (t_{(i\pm1)}(e_i = PART, \nu), B)$. A pool ball in sectors $(i+2)$ or $(i-2)$ must have a partition event in order to reach sectors $(i+1)$ or $(i-1)$ respectively followed by the unobstructed traversal of sectors $(i+1)$ or $(i-1)$ to reach sector i. Similarly, a pool ball in sector $(i+n)$ or $(i-n)$ must have a partition event in sector $(i+n)$ or $(i-n)$ followed by traversing $(n-1)$ sectors to reach sector i. This results in equation 27 of Lemma 5.2.

Lemma 5.3: The time of the earliest transient ball message which can be received by sector i, $u_i(\nu)$, is not solvable for the pool balls simulation.

Proof: If the next event E_i is not a partition event, then $t_i(e_i, \nu) \leq t_i(e_i = PART, \nu')$ where $\nu' > \nu$ by definition of monotonicity of events. The ball in sector i which has the event E_i is in the set of pool balls $G_i(\nu)$. The set of pool balls $G_i(\nu)$ cannot be propagated in time to calculate $E_i = (t_i(e_i = PART, \nu'), B)$ because the set of pool balls $G_i(\nu')$ cannot be determined. The set $G_i(\nu')$ cannot be determined because during the time interval corresponding to $(\nu' - \nu)$, additional pool balls may arrive in sector i from sectors $(i \pm 1)$. By definition of Wieland's sectoring strategy, sector i knows only of the existence of pool balls in i and not of any other sector; therefore, it is impossible for sector i to predict the arrival of additional pool balls in the interval $(\nu' - \nu)$. Since this is true, it is impossible for sector i to predict the event $E_i = (t_i(e_i = PART, \nu'), B)$. Without knowing E_i, the values $t_i(e_i = PART, \nu')$ cannot be determined to solve equation 27. Without B in E_i, it is impossible to determine $V_X(B)$ and therefore it is impossible to determine $TTC_i(e_i = PART, \nu')$ from Definition 5.7. This is true not only for sector i but for all sectors in {0,1,2...}; therefore, $u_i(\nu)$ is not solvable.

Definition 5.8: For any sector i and any iteration ν, the set of pool balls $G_i(\nu)$ has a pool ball whose X-axis velocity is greater than or equal to all of the other balls in $G_i(\nu)$. From Definition 5.7, this ball will have the minimum time to cross a sector. Denote this *minimum time to cross* by $TTC_{i_{min}}$; then,

$$TTC_{i_{min}}(\nu) \triangleq \min(TTC_i(e_i, \nu)) \tag{28}$$

for all pool balls in $G_i(\nu)$.

Definition 5.9: The *estimator of* $u_i(\nu)$, denoted $MST_i^1(\nu)$ is defined as

$$MST_i^1(\nu) \triangleq \min\left(t_{i\pm n}(e_{i\pm n}, \nu) + (n-1) * TTC_{i\pm n_{min}}(\nu)\right) \qquad (29)$$
$$\forall n : n \neq 0,\ n - i \geq 0,\ n + i \leq P$$

where $i \in \{0,1,2,\ldots,(P-1)\}$, and P is the number of sectors specified by the user.

Theorem 5.1: The estimator $MST_i^1(\nu)$ is valid.

Proof: Any estimator of $u_i(\nu)$ which satisfies Lemma 5.1 is valid and simulation correctness is therefore guaranteed. Lemma 5.1 can be shown to be satisfied as follows. The next event E_i has an event time which is always greater than zero by definition of monotonicity of events. The maximum X-axis velocity of any pool ball in $G_i(\nu)$ is always less than infinity; therefore, the minimum time to cross any sector, $TTC_{i_{min}}(\nu)$, is always greater than zero. Therefore, $MST_i^1(\nu) > 0$ for all i in {0,1,2...}.

If $e_i \neq PART$ then $t_i(e_i \neq PART, \nu) < t_i(e_i = PART, \nu')$ by definition of monotonicity. From Definition 5.8, $TTC_{i_{min}}(\nu) \leq TTC_i(e_i = PART, \nu)$. Therefore, $MST_i^1(\nu) \leq u_i(\nu)$. Lemma 5.1 is satisfied; therefore, $MST_i^1(\nu)$ is valid.

Theorem 5.2: The estimator $MST_i^1(\nu)$ guarantees progress; that is, there exists a sector whose event E_i can be executed for all ν.

Proof: Given P sectors where P is specified by the user, there exists a sector i, $i \in \{0,1,2\ldots P-1\}$, whose next event time is less than or equal to the next event time for all other sectors. From Definition 5.9, the minimum safe time estimator is the minimum of all other sector's next event times plus some overhead. Therefore, sector i's minimum safe time is at best the minimum of the remaining event times, but sector i's event time is less than or equal to all of the others; therefore, sector i's next event time must be less than or equal to its minimum safe time estimate. From Definition 5.5, sector i can execute. This is true for all ν; therefore, progress is guaranteed.

5.3.1 Additional Properties of $MST_i^1(\nu)$ Two interesting properties of $MST_i^1(\nu)$ can be shown. First, no two adjacent sectors can execute an event simultaneously if their event times are not equal. Second, the maximum number of sectors which *can* execute simultaneously is $\lceil \frac{P}{2} \rceil$ where P is the number of sectors specified by the user. This has implications for sector assignments which will be discussed later.

Lemma 5.4: Adjacent sectors cannot both meet their minimum safe times using $MST_i^1(\nu)$ as an estimator of $u_i(\nu)$ if their event times are not equal.

Proof: Let two adjacent sectors be denoted by (i) and $(i+1)$. If $t_i(e_i,\nu) \neq t_{(i+1)}(e_{(i+1)},\nu)$ then either $t_i(e_i,\nu) > t_{(i+1)}(e_{(i+1)},\nu)$ or $t_{(i+1)}(e_{(i+1)},\nu) > t_i(e_i,\nu)$ by definition. Both cases can be shown to have the property that at least one of the two sectors cannot execute their event.

CASE 1: $t_i(e_i,\nu) > t_{(i+1)}(e_{(i+1)}\nu)$
From Theorem 5.1, $MST_i^1(\nu)$ is the minimum of all other sector's event times plus some overhead. For adjacent sectors, the value of n in equation 29 is one; therefore, the additive term for the time to cross is zero. Thus, when comparing adjacent sectors only, the minimum safe time of sector i is the minimum of its two neighbor's next event times. Therefore, $MST_i^1(\nu)$ is at most equal to $t_{(i+1)}(e_{(i+1)},\nu)$ such that $MST_i^1(\nu) \leq t_{(i+1)}(e_{(i+1)},\nu)$. If $t_i(e_i,\nu)$ is greater than $t_{(i+1)}(e_{(i+1)},\nu)$ then $t_i(e_i,\nu) > MST_i^1(\nu)$ and execution is not allowed by Definition 5.5.

CASE 2: $t_{(i+1)} > t_i(e_i,\nu)$
The argument of case 1 is the same for case 2 resulting in $MST_{(i+1)}^1(\nu) \leq t_i(e_i,\nu)$ and $t_{(i+1)}(e_{(i+1)},\nu) > t_i(e_i,\nu)$; therefore, $t_{(i+1)}(e_{(i+1)},\nu) > MST_i^1(\nu)$ and execution is not allowed by Definition 5.5. Therefore, for both cases, *at least one* sector of two adjacent sectors cannot execute their next event if their next event times are not equal.

Lemma 5.5: If each sector i in {0,1,2...P-1} has a unique next event time, then the maximum number of sectors that can execute their event in parallel is $\lceil \frac{P}{2} \rceil$ where P is the number of sectors specified by the user.

Proof: From Lemma 5.4, no two adjacent sectors can execute if their event times are unique. Since the pool table is partitioned in a linear array, the maximum number of non-adjacent sectors is $\lceil \frac{P}{2} \rceil$.

5.3.2 Analysis of $MST_i^1(\nu)$ It is clear from Definition 5.9 that every sector must have knowledge of every other sector's candidate next event and minimum time to cross. Global communications are thus required. Several techniques are available to accomplish this. The simplest technique is to have each sector broadcast its next event time and maximum velocity to every other sector. Given M nodes, this technique requires $O(M^2)$ communications time, and each message requires more than two orders of magnitude of time to process than floating division (1). To reduce the number of communications, each node could communicate in a logarithmic fashion thereby requiring only $O(M \log M)$ communications time. The lower bound on communications is $O(M)$ if each node communicates only with its immediate neighbors (in terms of a pool table sector neighbor, not a hypercube node neighbor). Each of the techniques requires every node to wait for the slowest node. After each sector determines its candidate next event, each sector must calculate its MST. To do this, each sector must know every other sector's candidate next event time and every other sector's fastest ball velocity (X-axis only). The node which finishes calculating its candidate next event first must necessarily wait to calculate its MST until the last node calculates its candidate next event. A broadcast communications scheme cannot eliminate the potential wait state. Therefore, the $O(M)$ communcations scheme is the optimum implementation. This scheme requires an end sector to communicate with its immediate neighbor. In this manner, sector 0 sends its data to sector 1. Sector 1 combines its data with that of sector 0 and sends a single message to sector 2. Finally, sector (P-1) receives a message from sector (P-2) thereby giving sector (P-1) all of the data from every other sector. This data can then be passed back to each sector. Parallelism can be achieved by recognizing the independance of sector 0 and sector (P-1). As sector 0 sends its data to sector 1, sector (P-1) can send its data to sector (P-2) *in parallel.* This technique performs in lockstep and requires every sector to wait on the two end sectors.

5.4 Developing an Alternative MST Calculation

Chandy and Misra's paradigm requires the following constraints:

1. LP_i sends LP_j a message if and only if PP_i has an edge connecting PP_j (6:443).

2. There exists a prespecified constant ϵ such that
   $t_k - t_{k-1} > \epsilon$ for $k = 2, \cdots, k$ (6:442).
3. The minimum safe time for sector i is the minimum of the message tuples (t_k, m_k) received from all input arcs (T_{ik}) (6:444).

The MST calculation stated in Definition 5.9 is slightly different from that proposed by Chandy and Misra. Specifically, items one and three above disallow global communications. Chandy and Misra require an MST based only upon the input arcs represented by the edges connecting each PP_i. In the strictest sence, Definition 5.9 does not adhere to a true Chandy-Misra paradigm. It is desirable to develop an estimator that conforms to Chandy and Misra because their paradigm avoids global communications. This factor allows their paradigm to be scalable across any cube size where as $MST_i^1(\nu)$ is not scalable. As the number of nodes increases, the communications overhead of $MST_i^1(\nu)$ can be expected to negate any gains from potential parallelism.

5.5 Developing a Second Minimum Safe Time Calculation

This section develops the estimator $MST_i^2(\nu)$ which will be shown to be valid but does not guarantee progress. The estimator is derived from $MST_i^1(\nu)$.

Definition 5.10: There exists an upper bound on the velocity for any pool ball. Due to the conservation of energy and momentum, the total energy of all of the pool balls will not change after initialization. Therefore, there is a maximum velocity that any pool ball can have in the X-axis direction based upon the initialization values and there is a corresponding *global minimum time to cross* any sector. The absolute minimum time to cross a sector for any pool ball once initialized is denoted TTC_{global_min}.

Definition 5.11: The estimator of $u_i(\nu)$, denoted $MST_i^2(\nu)$, is defined as

$$MST_i^2(\nu) \triangleq \min\left(t_{i\pm1}(e_{i\pm1}, \nu),\, TTC_{global\_min}\right) \tag{30}$$

Theorem 5.3: The estimator $MST_i^2(\nu)$ is valid.

Proof: Any estimator of $u_i(\nu)$ which satisfies Lemma 5.1 is valid. By definition,

$$\begin{aligned} MST_i^1(\nu) &= \min\left(t_{i\pm n}(e_{i\pm n},\nu) + (n-1) * TTC_{i\pm n_{min}}(\nu)\right) \\ &\quad \forall n:\ n \neq 0,\ n-i \geq 0,\ n+i < P \\ &= \min\left(\begin{array}{l} t_{i\pm 1}(e_{i\pm 1},\nu), \\ t_{i\pm n}(e_{i\pm n},\nu) + (n-1) * TTC_{i\pm n_{min}}(\nu) \end{array}\right) \\ &\quad \forall n:\ n \neq 0,\ n \neq 1,\ n-i \geq 0,\ n+i < P \end{aligned}$$

Also,

$$\begin{aligned} 0 + (n-1) * TTC_{i\pm n_{min}}(\nu) &< t_{i\pm n}(e_{i\pm n},\nu) + (n-1) * TTC_{i\pm n_{min}}(\nu) \qquad (31) \\ \text{if } t_{i\pm n}(e_{i\pm n},\nu) &> 0 \end{aligned}$$

The next event time is always greater than zero by definition of monotonicity of events; therefore, the inequality of equation 31 is true. Furthermore,

$$\begin{aligned} (n-1) * TTC_{global\_min} &\leq (n-1) * TTC_{i\pm n_{min}}(\nu) \qquad (32) \\ \text{if } TTC_{global\_min} &\leq TTC_{i\pm n_{min}} \end{aligned}$$

The global minimum time to cross is always less than or equal to $TTC_{i\pm n_{min}}$ by Definition 5.10; therefore, the inequality of equation 32 is true. Last,

$$TTC_{global\_min} \leq (n-1) * TTC_{global\_min}$$

These substitutions reduce $MST_i^1(\nu)$ to $MST_i^2(\nu)$ and $MST_i^2(\nu) < MST_i^1(\nu)$. Since $MST_i^1(\nu)$ has been proven valid, then by Lemma 5.1, $MST_i^2(\nu)$ is valid.

Theorem 5.4: The estimator $MST_i^2(\nu)$ does not guarantee progress.

Proof: Given that $MST_i^2(\nu) = \min\left(t_{(i\pm 1)}(e_{(i\pm 1)},\nu),\ TTC_{global\_min}\right)$ from Definition 5.11, suppose that $TTC_{global\_min} < t_i(e_i,\nu)$ for all i in {0,1,2...P-1}. Then $MST_i^2(\nu) = TTC_{global\_min}$ for all i by definition of $MST_i^2(\nu)$. If the minimum safe time is less than

the event time for every sector i, then no sector can execute its next event by Definition 5.5. The value of TTC_{global_min} is a constant by definition of conservation of energy and momentum; therefore, for all values of ν, $MST_i^2(\nu) = TTC_{global\_min}$. The next event times for all sectors will remain unchanged because each value of ν produces the same results if none of the pool balls ever change state information. For this example, the simulation will run indefinitely while the simulation time will remain at zero!

5.6 Developing a Third MST Calculation

The third MST formulation relies heavily upon Chandy and Misra's concept of *input arcs* and *output arcs*. A sector i outputs a message to a neighboring sector which represents the earliest estimated time that sector i can send a transient ball message to its neighbor. This is an output arc from sector i to sector $(i \pm 1)$. Equivalently, each sector receives a message sent from its neighbors as input arcs. The formulation for the third MST estimator which adheres to Chandy and Misra's constraints has the following logic. If one iteration ago at $\nu = (\nu - 1)$, sector $(i + 1)$ passed a message to sector i indicating that no transient ball messages will be sent before time t_1 then for the next iteration ν, it must still be true that sector $(i + 1)$ will not send a transient ball message before time t_1 due to monotonicity of events. Furthermore, if a pool ball were to cross sector $(i + 1)$ into sector i in no less than time TTC_{global_min}, and one iteration ago at $(\nu - 1)$ sector $(i + 1)$ could not output a transient ball message until at least time t_1, then for iteration ν, sector $(i + 1)$ cannot output a transient ball message to sector i until at least time $t_2 = t_1 + TTC_{global\_min}$. This concept allows the MST estimator to constantly increase in size until at least one sector can execute an event. After executing the event, there is no guarantee that an event can be executed for iteration $(\nu + 1)$, but there is a guarantee that execution will be possible before $(\nu + \infty)$ because the estimator itself constantly increases with ν. The potential to have non-executing iterations reduces the efficiency of the parallel simulation and the lower estimate of the MST reduces the probability of multiple executing sectors; however, the paradigm is scalable to M nodes where M is limited only by the hardware. The following definitions support the development of the Chandy and Misra estimator and the theorems that follow.

Definition 5.12: Let $O_{i,(i+1)}(\nu)$ be the *output arc* of sector i to sector $(i+1)$ at some discrete time interval ν. Then,

$$\begin{aligned} O_{i,(i+1)}(\nu) &\triangleq \min\left(t_i(e_i,\nu),\ I_{i,(i-1)}(\nu-1) + TTC_{global\_min}\right) \\ O_{i,(i-1)}(\nu) &\triangleq \min\left(t_i(e_i,\nu),\ I_{i,(i+1)}(\nu-1) + TTC_{global\_min}\right) \end{aligned} \tag{33}$$

where $I_{i,(i\pm1)}(\nu-1)$ is the *input arc* to sector i from sector $(i\pm1)$. The inputs and outputs are related by $I_{i,(i\pm1)}(\nu) = O_{(i\pm1),i}(\nu)$ and all inputs and outputs at $(\nu=0) = 0$.

Definition 5.13: The *estimator of* $u_i(\nu)$, denoted $MST_i^3(\nu)$, is defined as

$$MST_i^3(\nu) \triangleq \min\left(I_{i,(i+1)}(\nu),\ I_{i,(i-1)}(\nu)\right) \tag{34}$$

Theorem 5.5: The estimator $MST_i^3(\nu)$ is valid.

Proof: To prove that $MST_i^3(\nu)$ is valid, it will be shown that for all ν, $MST_i^3(\nu) \leq MST_i^1(\nu)$. This will be done by analyzing the specific instances for $\nu = 1$ and 2 from which a general solution for all ν clearly presents itself. The general solution is equivalent to the combination of Definitions 5.12 and 5.13.

CASE 1: $\nu = 1$

Through substitution, $MST_i^3(\nu=1)$ can be equivalently written as $\min\left(O_{(i\pm1),i}(1)\right)$. Substituting the output terms with Definition 5.12 results in the following equation:

$$\begin{aligned} MST_i^3(1) &= \min\begin{pmatrix} \min(t_{i+1}(e_{i+1},1),\ TTC_{global\_min}), \\ \min(t_{i-1}(e_{i-1},1),\ TTC_{global\_min}) \end{pmatrix} \\ &= \min\begin{pmatrix} t_{i\pm1}(e_{i\pm1},1) + 0 * TTC_{global\_min}, \\ t_{i\pm2}(e_{i\pm2},0) + 1 * TTC_{global\_min} \end{pmatrix} \end{aligned}$$

where $t_i(e_i, \nu=0) = 0$ for all i.

CASE 2: $\nu = 2$

Using the same substitution steps, the following is seen to be true for $\nu = 2$.

$$
\begin{aligned}
MST_i^3(2) &= \min \begin{pmatrix} \min(t_{i+1}(e_{i+1},2),\ I_{(i+1),(i+2)}(1) + TTC_{global\_min}) \\ \min(t_{i-1}(e_{i-1},2),\ I_{(i-1),(i-2)}(1) + TTC_{global\_min}) \end{pmatrix} \\
&= \min \begin{pmatrix} \min(t_{i+1}(e_{i+1},2),\ O_{(i+2),(i+1)}(1) + TTC_{global\_min}) \\ \min(t_{i-1}(e_{i-1},2),\ O_{(i-2),(i-1)}(1) + TTC_{global\_min}) \end{pmatrix} \\
&= \min \begin{pmatrix} \min \begin{bmatrix} \min(t_{i+2}(e_{i+2},1),\ TTC_{global\_min}) \\ t_{i+1}(e_{i+1},2) \end{bmatrix} + TTC_{global\_min} \\ \\ \min \begin{bmatrix} \min(t_{i-2}(e_{i-2},1),\ TTC_{global\_min}) \\ t_{i-1}(e_{i-1},2) \end{bmatrix} + TTC_{global\_min} \end{pmatrix} \\
&= \min \begin{pmatrix} t_{i\pm1}(e_{i\pm1},2) + 0 * TTC_{global\_min}, \\ t_{i\pm2}(e_{i\pm2},1) + 1 * TTC_{global\_min}, \\ t_{i\pm3}(e_{i\pm3},0) + 2 * TTC_{global\_min} \end{pmatrix}
\end{aligned}
$$

where $t_{i\pm n}(e_{i\pm n}, \nu = 0) = 0$.

GENERAL SOLUTION:

From the preceding two cases, the general solution for all ν is:

$$MST_i^3(\nu) = \min(t_{i\pm n}(e_{i\pm n},(\nu - n + 1)) + (n-1) * TTC_{global\_min}) \qquad (35)$$

Equation 35 has a striking resemblance to the definition of $MST_i^1(\nu)$; however, it is easily shown that $MST_i^3(\nu) \leq MST_i^1(\nu)$ as follows. For all i, $TTC_{global\_min} \leq TTC_{i_{min}}$ from Definition 5.10. Due to monotonicity of events, $t_{i\pm n}(e_{i\pm n},(\nu-n+1)) \leq t_{i\pm n}(e_{i\pm n},\nu)$ since $(\nu - n + 1) \leq \nu$ for $(n > 1)$. Therefore, $MST_i^3(\nu) \leq MST_i^1(\nu)$ which implies that $MST_i^3(\nu)$ is valid.

Theorem 5.6: The estimator of $u_i(\nu)$, $MST_i^3(\nu)$, guarantees progress.

Proof: After substituting the definition of output arcs into the definition of $MST_i^3(\nu)$, the estimator can be written as

$$\begin{aligned} MST_i^3(\nu) &= \min\left(O_{(i+1),i}(\nu), O_{(i-1),i}(\nu)\right) \\ &= \min\begin{pmatrix} \min\left[t_{i+1}(e_{i+1},\nu), O_{(i+2),(i+1)}(\nu) + TTC_{global\_min}\right] \\ \min\left[t_{i-1}(e_{i-1},\nu), O_{(i-2),(i-1)}(\nu) + TTC_{global\_min}\right] \end{pmatrix} \end{aligned}$$

This equation clearly shows that the value of $MST_i^3(\nu)$ constantly increases because either the next event time $t_{i\pm1}(e_{i\pm1},\nu)$ increases by definition of monotonicity of events, or the term representing the output arc of previous iterations increases by a constant factor TTC_{global_min}. Therefore, execution of events is possible as ν increases.

5.7 Analyzing Alternative MST Calculations

$MST_i^1(\nu)$ requires global communications whereas $MST_i^3(\nu)$ does not. The communications time for $MST_i^1(\nu)$ can be reduced to at most $O(M)$ where M is the number of nodes. The communications time for $MST_i^3(\nu)$ is $O(1)$. Thus, scalability is a major tradeoff. $MST_i^1(\nu)$ specifically requires $(M-1)$ communications per node while $MST_i^3(\nu)$ requires a constant two communications per node for all but the end sectors which require only one. Thus, for $M = 2$, the two MST calculations require approximately the same communications time. For $M = 4$, $MST_i^3(\nu)$ is perhaps slightly superior *if all else is the same*. Clearly, all else is not the same because $MST_i^1(\nu)$ guarantees to execute at least one event per iteration whereas $MST_i^3(\nu)$ does not. This factor constitutes the tradeoff between execution rate and communications time complexity. Not only can $MST_i^1(\nu)$ guarantee to execute at least one event per iteration, but up to $\lceil \frac{P}{2} \rceil$ sectors can execute per iteration. A simple four sector example highlights the differences between the two paradigms.

Conjecture 5.1 For small M where M is the number of nodes in the pool balls simulation, $MST_i^1(\nu)$ is superior to $MST_i^3(\nu)$. The size of M has been shown to be at least 4.

5.7.1 An Example using Both MST's To demonstrate the implementation of both conservative paradigms, consider the four node, four sector pool balls simulation diagram

of Figure 12. The terms 'NET' and 'TTC' represent the next event time and time to cross a sector for the fastest ball in sector i. The simulation time for each sector is currently 0.00 seconds.

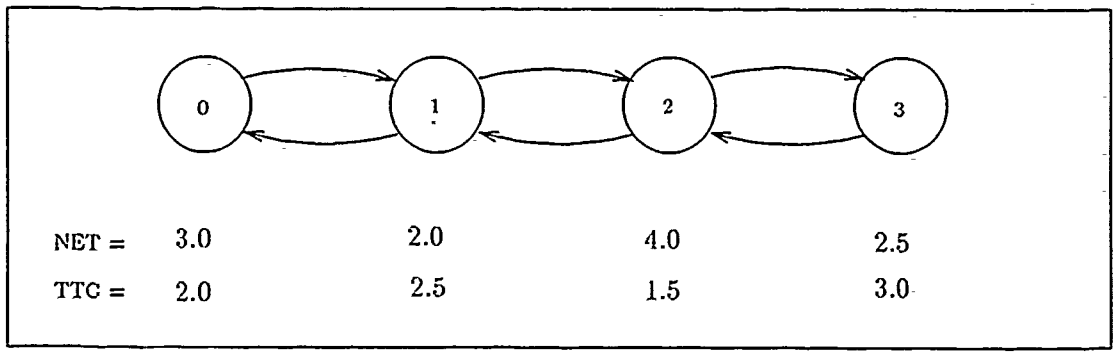


Figure 12. A Four Node, Four Sector Process Graph

Using the $MST_i^1(\nu)$ with Global Communications

Each message sent from sector i to sector k has the form $(NET_i, TTC_i, NULL)$.

Node 0 sends (3.0, 2.0, $NULL$) to Node $< 1,2,3 >$
Node 1 sends (2.0, 2.5, $NULL$) to Node $< 0,2,3 >$
Node 2 sends (4.0, 1.5, $NULL$) to Node $< 0,1,3 >$
Node 3 sends (2.5, 3.0, $NULL$) to Node $< 0,1,2 >$

$$MST_0 = \min\{(2.0+0*2.5), (4.0+1*1.5), (2.5+2*3.0)\} = 2.0$$
$$MST_1 = \min\{(3.0+0*2.0), (4.0+0*1.5), (2.5+1*3.0)\} = 3.0$$
$$MST_2 = \min\{(3.0+1*2.0), (2.0+0*2.5), (2.5+0*3.0)\} = 2.0$$
$$MST_3 = \min\{(3.0+2*2.0), (2.0+1*2.5), (4.0+0.1.5)\} = 4.0$$

$\therefore$ $\overline{P(E_0)}, P(E_1), \overline{P(E_2)}, P(E_3)$

Using the $MST_i^3(\nu)$ with Constant Communications

For $MST_i^3(\nu)$, the global minimum TTC must be known a priori. For the scenario of Figure 12, the minimum TTC is 1.5 for all sectors.

LOOP 1

$O_{0,1}(1) = \min(3.0, 1.5) = 1.5$

$O_{1,2}(1) = \min(2.0, 1.5) = 1.5$

$O_{2,3}(1) = \min(4.0, 1.5) = 1.5$

$O_{3,2}(1) = \min(2.5, 1.5) = 1.5$

$O_{2,1}(1) = \min(4.0, 1.5) = 1.5$

$O_{1,0}(1) = \min(2.0, 1.5) = 1.5$

$MST_0(1) = \min(1.5) = 1.5$

$MST_1(1) = \min(1.5, 1.5) = 1.5$

$MST_2(1) = \min(1.5, 1.5) = 1.5$

$MST_3(1) = \min(1.5, 1.5) = 1.5$

$\therefore \overline{P(E_0)}, \overline{P(E_1)}, \overline{P(E_2)}, \overline{P(E_3)}$

At this point, the first high level loop construct has finished. Each node calculated a candidate next event, determined its MST and executed its next event for all sectors provided $NET \leq MST$. This example shows that none of the sectors could safely execute an event based upon the information provided. The estimator $MST_i^1(\nu)$ executed two events on two nodes thereby achieving 50% parallelism during the execution phase. This was possible because each node had additional information upon which to calculate a superior MST at the cost of increased communications. The estimator $MST_i^3(\nu)$ must implement a second high level loop to attain the same simulation state shown as follows.

LOOP 2

$O_{0,1}(2) = \min(3.0, (1.5 + 1.5)) = 3.0$

$O_{1,2}(2) = \min(2.0, (1.5 + 1.5)) = 2.0$

$O_{2,3}(2) = \min(4.0, (1.5 + 1.5)) = 3.0$

$O_{3,2}(2) = \min(2.5, (1.5 + 1.5)) = 2.5$

$O_{2,1}(2) = \min(4.0, (1.5 + 1.5)) = 3.0$

$O_{1,0}(2) = \min(2.0, (1.5 + 1.5)) = 2.0$

$MST_0(2) = \min(2.0) = 2.0$
$MST_1(2) = \min(3.0, 3.0) = 3.0$
$MST_2(2) = \min(2.0, 2.5) = 2.0$
$MST_3(2) = \min(3.0) = 3.0$

$\therefore \overline{P(E_0)}, P(E_1), \overline{P(E_2)}, P(E_3)$

The scenario above illustrates the typical 'wind up' overhead of conservative simulations implemented with Chandy-Misra. The number of loops required to reach an executable state depends upon the difference between TTC_{min} and NET_{min}. The scenario presented incorporated a Δt small enough that the wind up cost consisted of only one loop; however, this will not always be the case. Even after the windup is finished, the proof presented earlier validates the possibility that $\forall k: MST_k < t_k$.

5.8 Selecting an MST Formulation

Both equations for calculating the minimum safe time were considered for this thesis effort. The estimator $MST_i^1(\nu)$ seemed to be more intrinsically programmable and AFIT is currently limited to an eight node hypercube which favors $MST_i^1(\nu)$ due to small cube size. No attempt was made to implement both strategies so empirical data is not available to date. The following sections describe the implementation strategy used to incorporate $MST_i^1(\nu)$.

5.8.1 Implementing the Minimum Safe Time Implementing Definition 3.2 is straight forward. Two message communications data structures were defined. Both data structures have the following fields:

1. Sector Number
2. Candidate Next Event Time
3. Time to Cross (for the fastest ball in Sector i)
4. Time to be affected by any left sector.

5. Time to be affected by any right sector.

6. MST_i

Sector zero completes fields one through four. Field four is *infinity* (defined in a header file to be 99999999.99 seconds) since sector 0 has no left neighbor. This structure is next passed to sector 1. Sector 1 fills in fields one through four. This structure is then passed to sector 2. This series continues until sector $(P-1)$ receives the data structure. At this point, every sector $(0-K)$ knows the earliest that it can receive a transient ball message from any sector to their left. As this entire process takes place, sector *(P-1)* sends a data structure to its neighbor *(P-2) in parallel.* Sector *(P-1)* assigns infinity to the time to be affected by any right sector as it has no right neighbor. Assuming near equal processing time, sector *(P-1)* will receive its message originated by sector 0 at the same time that sector 0 receives its message originated by sector *(P-1)*. At this point, every sector now knows the earliest time at which they can receive a transient ball message from either the left or the right. The *MST* is simply the minimum of these two values.

5.8.2 Implementing Wieland's Data Replication Strategy The data replication strategy remains basically unchanged from the partitioned sequential version. Pool ball objects must still transition from one sector to an adjacent sector in a two step process via PARTITION events and EXIT events. If adjacent sectors are collocated on one node, data replication may take place directly through memory as described in Section 4.3.4. If $sector_i$'s adjacent sector resides on a different node, the rules for data replication presented in Section 4.3.4 must be enforced through discrete message passing. It is imperative that each sector have its data replication updated prior to determining its next event; otherwise, incorrect events can occur. For example, $sector_i$ could have a replicated ball object stored in its ball object manager. This replicated ball could be scheduled for a collision with another of $sector_i$'s ball objects. If the replicated ball was previously moved by the owning sector (i.e. $sector_{i-1}$ or $sector_{i+1}$), and if $sector_i$ had not updated its replicated copy, the predicted collision event would be in error. In fact, the case could arise that the replicated ball should not even be visible to $sector_i$ had the update been enforced. This condition requires that every sector wait to determine candidate events until data replication has been completed.

To implement the synchronous waiting condition, every node sends a message counter to the adjacent node(s) stating the number of data replications that will occur. If a node has no ball objects to send, the message counter sent equals zero. In this fashion, each

node can simultaneously execute scheduled events immediately followed by sending data replication message counters to adjacent nodes. At this point, lockstep synchronization is again enforced as no node may continue processing until it has received message counters from all adjacent nodes. Once a node receives a message counter from an adjacent node, that node is able to receive the proper number of replicated object commands.

5.9 Summary

The design of the pool balls simulation is object oriented. The equations of motion conform to the laws of physics for elastic collisions (i.e. conservation of energy and momentum) and frictionless motion. The parallel design of the pool balls simulation incorporated Schruben's concept of resident entities thereby modeling the table as a distributed set of table sectors. This has shown to reduce the amount of communications over a transient entity design approach. The paradigm developed for the synchronization of the distributed simulation has been shown to be conservative. This conservative paradigm is superior to that proposed by Chandy and Misra for small N. Both this paradigm and Chandy-Misra's paradigm avoids the possibility of deadlock via NULL message passing. With both paradigms, improved parallelism can be achieved by assigning multiple sectors (*LPs*) to individual nodes. The upper bound on the number of sectors that can safely execute a candidate event using the paradigm developed in this thesis is $\lceil \frac{P}{2} \rceil$; therefore, 100% parallelism is possible if and only if each node has assigned to it two or more table sectors.

VI. Test Results

6.1 Introduction

Chapter VI outlines the measures taken in this thesis effort to validate the pool balls software and the test procedures used to generate performance data. The results of the tests are discussed from which conclusions are made. The conclusions are stated in Chapter VII.

6.2 Verification and Validation

The pool balls simulation was designed and implemented in three major steps consisting of the following:

1. Design and implementation of a sequential simulation without spatial partitioning and data replication.
2. Design and implementation of a sequential simulation incorporating spatial partitioning and data replication.
3. Design and implementation of a parallel simulation incorporating spatial partitioning and data replication.

The first sequential simulation was validated in several stages consisting of the following tests:

1. Test a collision between a pool ball and a cushion (both horizontal and vertical).
   (a) Create a scenario with known behavior. Force the simulation to produce the specified pool balls (i.e. positions, times and velocities) and compare the simulation results with expected results.
   (b) Enable the simulation's random number generator to produce a random pool ball and collisions. Record the events to disk and verify output by calculating each event by hand.
2. Test a collision between two pool balls.
   (a) Create a scenario with known behavior. Force the simulation to produce the specified pool balls (i.e. positions, times and velocities) and compare the simulation results with expected results.

(b) Enable the simulation's random number generator to produce random pool balls and collisions. Record the events to disk and verify the output by calculating each event by hand.

Test 1a consisted of creating a single pool ball scenario on paper with known time, position and velocity. The random number generator was disabled to allow the creation of a pre-specified pool ball. Four separate tests were run to verify correct operation of a left and right vertical cushion collision and a top and bottom horizontal cushion collision. The simulation output was compared against the hand calculated results.

Test 1b consisted of creating a randomly generated pool ball and simulating 25 events. With only one pool ball, all 25 events were guaranteed to be limited to horizontal and vertical cushion events. The simulation output was checked by hand for all 25 events. This test was performed three times to produce a high level of confidence.

Test 2a consisted of creating various scenarios involving two or more pool balls pre-positioned to intentionally produce pool ball collisions at known times. The random number generator was disabled to allow creation of deterministic inputs. The simulation output was verified by comparing each collision (including the cushion collisions) with the expected hand calculated results.

Test 2b consisted of creating randomly generated pool balls and simulating 10 events. Each event was verified by hand. This test was performed five times with varying random inputs and number of pool balls.

Large quantities of pool balls as well as large quantities of events were not possible to test due to the labor intensive calculations required for comparison. While these limited tests do not prove system correctness, the test results produce a high level of confidence in system correctness.

The sequential simulation incorporating spatial partitioning and data replication was validated by comparing the output against the output of the first simulation under the following constraints:

1. The number of pool balls and initial conditions for both simulations were equal.
2. Partition and Exit events were not recorded to disk.
3. The user specified simulation time was the same for both simulations.

Since the pseudo random number generator uses a seed without reference to a system clock, each simulation test run produces the exact same initial conditions provided the seed remains unchanged. In this manner, the number of pool balls could be specified for both simulation software versions resulting in identical initial conditions. By not recording PARTITION and EXIT events to disk, the test outputs from both simulation versions should have been the same. This was verified by using the Unix 'diff' command on the two output files. To gain further confidence in valid system operation, sector crossings were verified by hand for 25 different pool balls corresponding to 25 different discrete points in simulation time.

The parallel simulation version was tested against the original sequential simulation and against the partitioned sequential version in that order. The first series of tests were identical to those discussed above. The Unix 'diff' command was used to highlight any differences in simulation outputs between the sequential, non-partitioned version and the parallel version. A lengthy test consisting of 100 pool balls and a simulation time of 60 seconds was used as a final test. During the second series of tests, all PARTITION and EXIT events were included in the simulation output. The sequential and parallel software versions were compared against each other to test the functionality of the border crossings. Again, 100 pool balls were simulated for 60 seconds. The Unix 'diff' command did not produce any differences between the two partitionable software versions.

6.3 Simulation Performance Test Plan

Several parameters were available to vary. It was desirable to gain insight into the performance of the implemented design as the parameters change. Scalability in terms of cube size and load factor performance are two qualities of particular interest. The following section defines the variables of interest which were scrutinized during the test phase.

6.3.1 Defining the Variables of Interest The number of nodes is a variable of interest without which speedup calculations are impossible. Therefore, all test cases defined must be duplicated for various node configurations. The software design imposed the constraint that the number of nodes selected must be a power of two. AFIT has an eight node hypercube; therefore, four test runs must be made for any given test case corresponding to one, two, four and eight nodes.

The number of pool balls is a variable of interest. Changing this variable allows inspection and analysis of the relationship between speedup and computational loading.

The algorithm design and analysis phase predicted that the speedup should generally improve with increased loading due to the overall $O(N^2)$ algorithm. Furthermore, as the loading increases, there is a greater probability that each sector will contain one or more pool balls and therefore produce useful candidate events which may be executed in parallel with other useful events.

The number of sectors is a variable of interest. The algorithm design and analysis phase predicted that the run time performance can improve with increased sectoring. If this is in fact true, then sectoring becomes crucial in the calculation of speedup. Since speedup relates the parallel run time to the *best* sequential time, the optimum sectoring on a single node must be determined. There was no technique available in the analysis phase to determine a priori the optimum sectoring for a given quantity of pool balls; therefore, sectoring *must* be a variable if the optimum sectoring is to be found.

6.3.2 Defining the Constants As variables of interest are changed from test to test, the simulation run time and pool table dimensions must remain constant to produce any meaningful results. The simulation run time was set to 2.00 seconds. This time was selected based upon the results of some trial experiments. Using 500 pool balls on a single node, the test run required approximately four hours of wall clock time. Using 10 pool balls, the test run required approximately one minute. This range seemed reasonable based upon the time constraints of this thesis effort.

The table dimensions were set to 1024 x 512 inches. The width of 512 was arbitrary. The length of 1024 was selected to provide a reasonable degree of sectoring capability. Given a one inch pool ball radius (arbitrarily selected), a length of 1024 inches allows up to 256 sectors of equal size such that no two sectors overlap and a pool ball can reside in a sector without overhanging into a border region between sectors.

6.3.3 The Test Plan The quantity of pool balls was tested at values of 10, 20, 30, 40, 50, 100 and 200. The range of sectors to implement was 1 to 68 based upon some sample test runs. The two, four and eight node tests varied the number of sectors by an even multiple of the number of nodes. The single node tests could have varied the number of sectors between 1 and 68 in multiples of one; however, multiples of two were arbitrarily selected to save time. After the initial series of tests were finished, additional tests were added. The performance curves were later extended by performing test runs at 300 and 400 pool balls. The quantities 120 and 160 were added to be able to compare test results

with those of Cal Tech. The results of the first seven tests are shown as figures 13 through 19.

6.4 Test Results

6.4.1 Analysis of Pool Table Sectoring Figures 13 through 19 graphically illustrate the effect of sectoring the pool table on one, two, four and eight nodes for various levels of computational loading. The execution time of figures 13 and 14 does decrease with increased sectoring. This is due to the fact that the gains from adding additional sectors are outweighed by the increase in intermediate events. Recall that the only real events of interest consist of collisions between pool balls and table cushions and collisions between different pool balls. A pool ball can traverse the entire table in one step using a single partition provided there are no pool ball collisions. Each additional sector adds two events (PARTITION and EXIT) for a pool ball to traverse the table. Hence, increasing the number of sectors increases the total number of events to process. Although the number of events to process increases, the time to determine the next event decreases with P, where P is the number of table sectors. As the number of sectors increases, the average number of pool balls in any sector decreases. If the average density decreases below one pool ball per sector, then there will be at least one sector which has no pool balls in it. Adding more sectors beyond this limit will therefore not decrease the search space; however, the total number of events to process will continue to increase. The tradeoff between decreasing the search space and increasing the total number of events determines the optimum number of sectors. When using multiple nodes, the tradeoff is less intuitive because of the effects described in Theorem 5.2. This theorem states that no two adjacent sectors can both meet their minimum safe times provided the next event times are not equal. Adding more sectors on multiple nodes therefore increases the probability that a node has at least one executable process from Theorem 5.3. The upper limit for increasing performance by adding additional sectors is still a density of one pool ball per sector since adding additional sectors beyond this point will not give a node a greater probability of executing a useful process.

Figure 18 has more of a parabolic shape. Notice that as the number of pool balls (loading factor) increases, sectoring the pool table increases in importance. Fig 19 appears to be more of a $\frac{1}{N}$ relationship; however, it is conjectured that the family of curves eventually rises with increased sectoring.

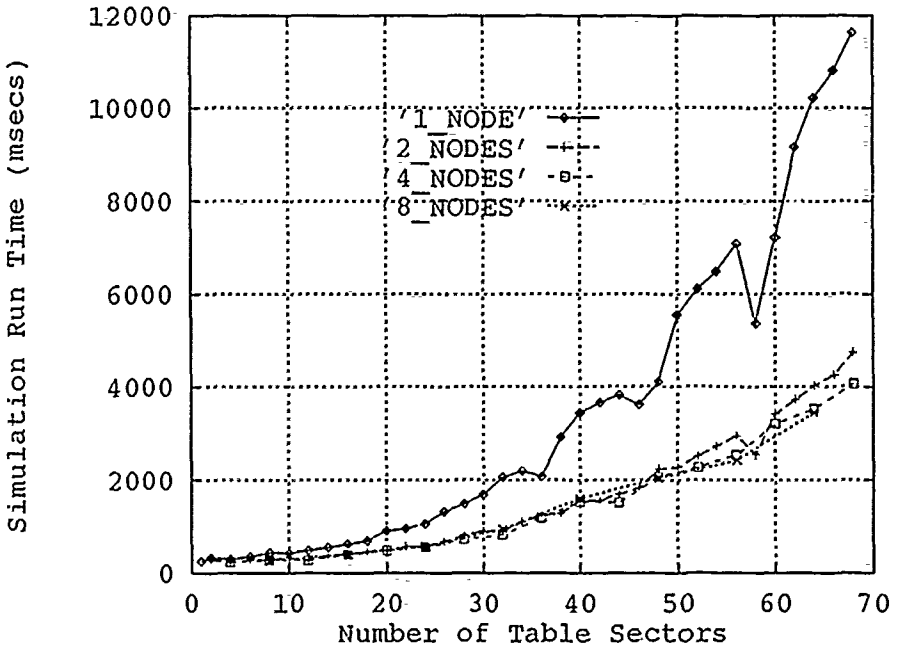


Figure 13. Performance Curves for 10 Balls

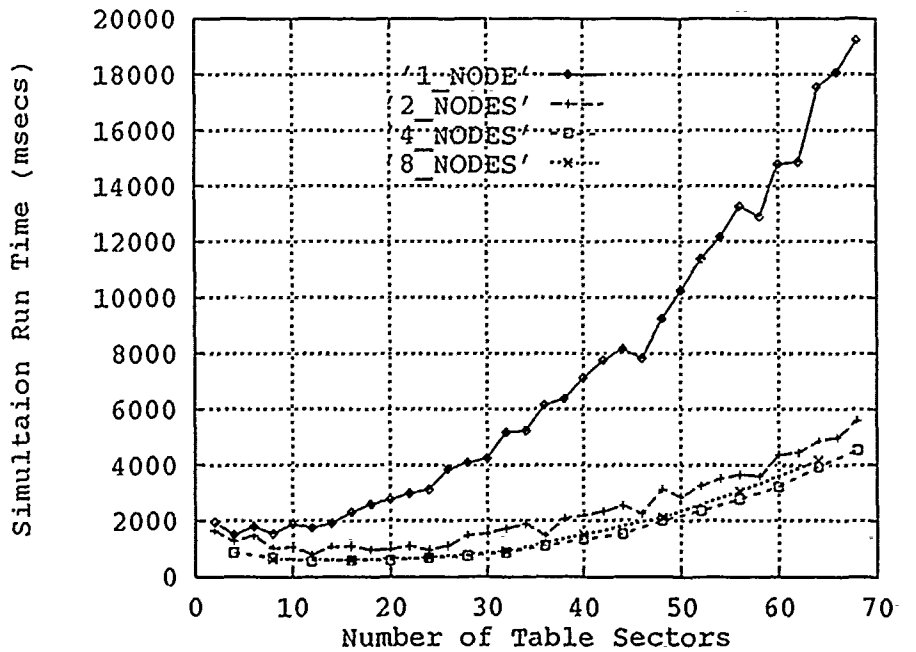


Figure 14. Performance Curves for 20 Balls

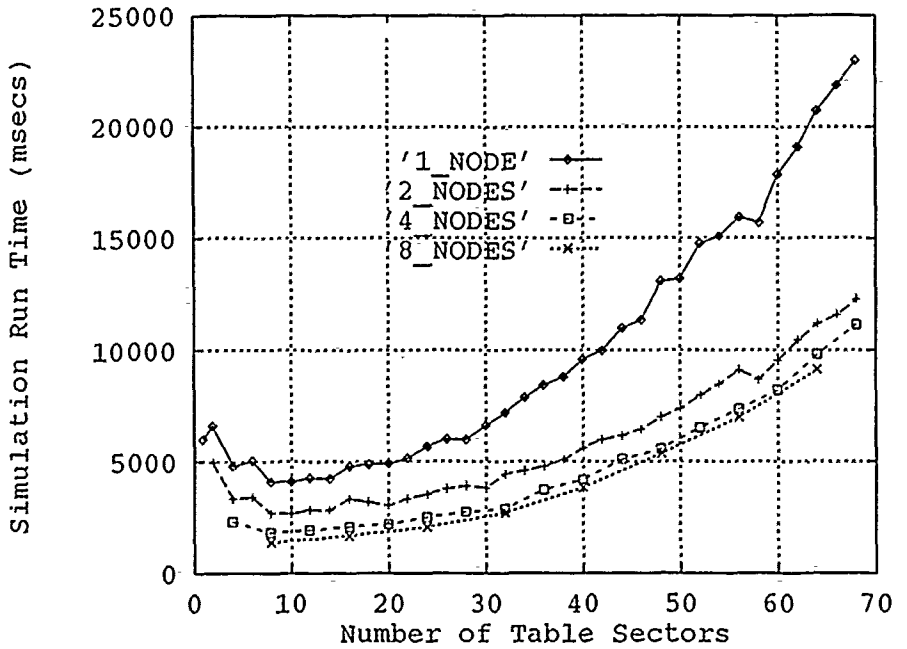


Figure 15. Performance Curves for 30 Balls

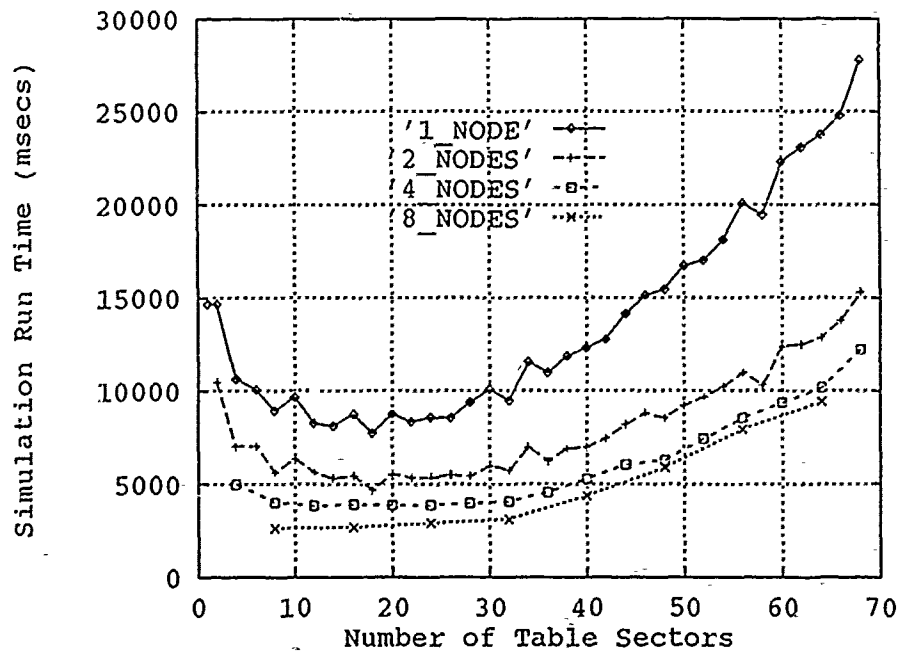


Figure 16. Performance Curves for 40 Balls

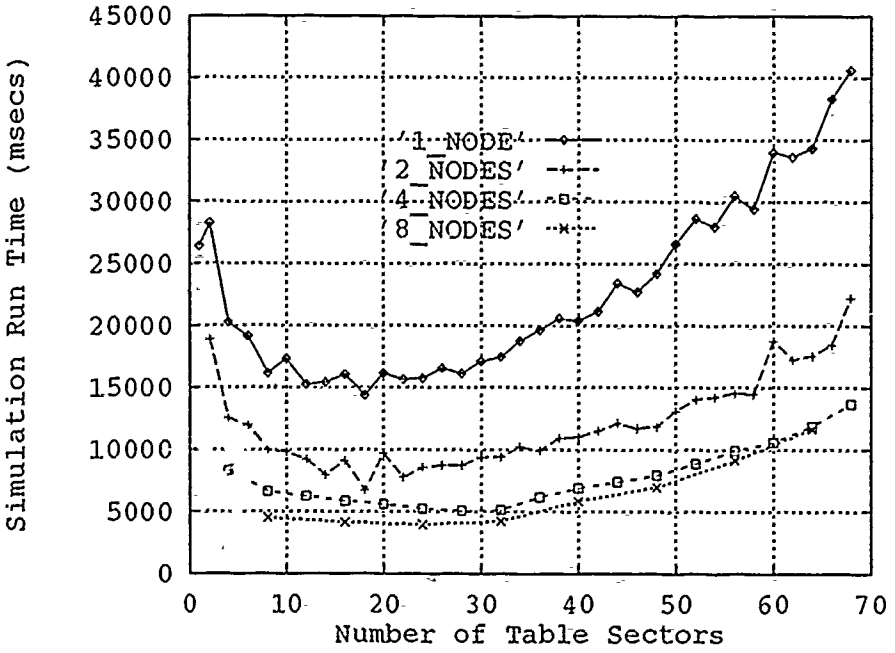


Figure 17. Performance Curves for 50 Balls

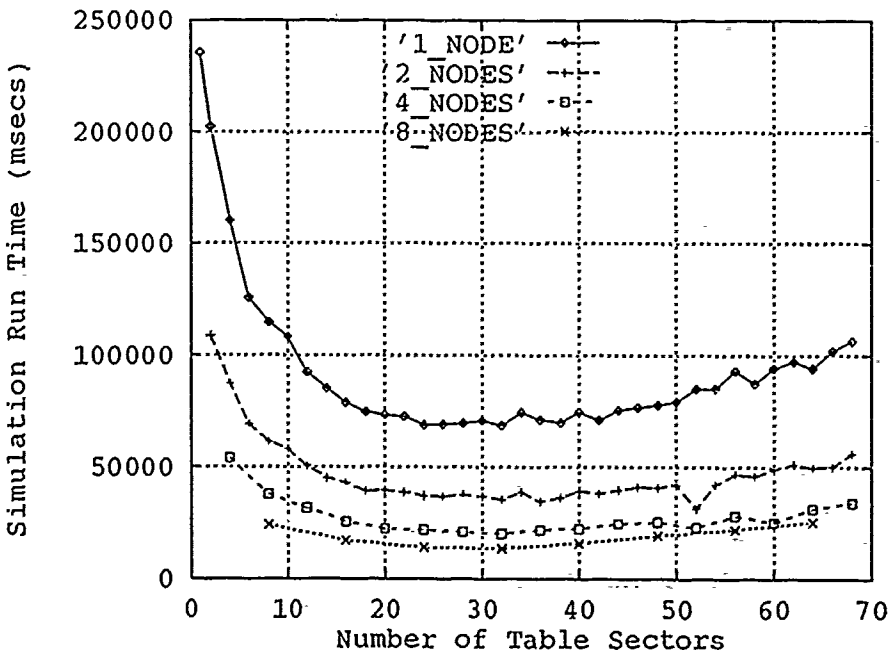


Figure 18. Performance Curves for 100 Balls

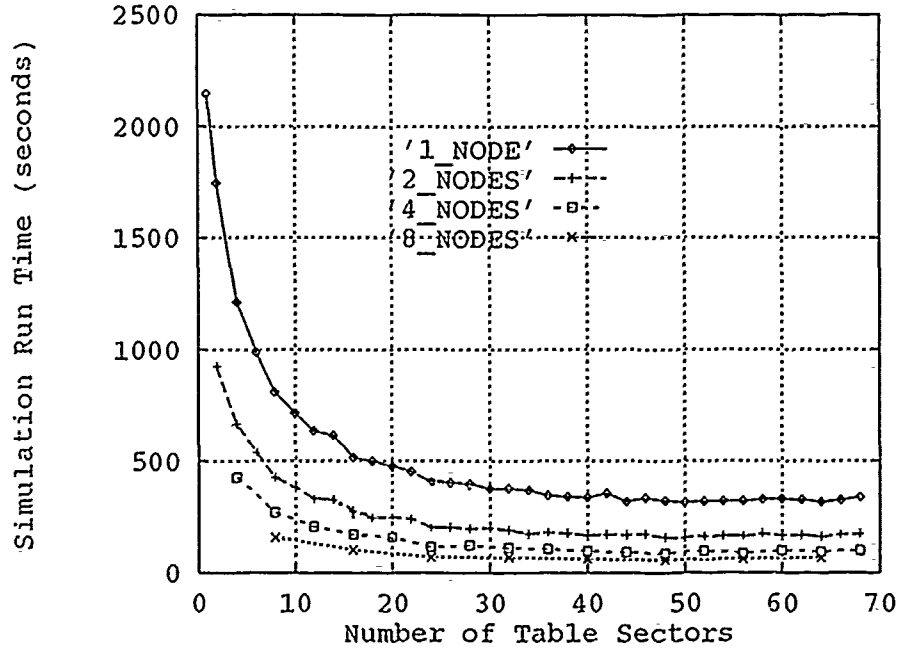


Figure 19. Performance Curves for 200 Balls

The curves of figure 20 indicate that the speedup for any cube size approaches an assymptotic limit regardless of load factor. This was surprising because it was expected that the speedup would generally improve with N. Since the algorithm time complexity is $O(N^2)$, increasing the number of pool balls should increase the computations to communications ratio thereby favoring the course granularity of the iPSC/2 hypercube. It is conjectured that this is not the case because the optimal number of sectors increases with N; therefore, the search space only increases with $O(\frac{N^2}{P})$. Furthermore, as the optimum number of sectors increase, the percentage of parallelism increases due to the effects of Lemma 5.4 and 5.5. The curves of figure 21 show the same data in the more traditional format.

The efficiency $(\frac{S}{N})$ is shown in figures 22 and 23. The curves of figure 22 show the relationship between efficiency and load factor. The efficiency appears to approach a limit for each cube size regardless of the number of objects to simulate. Both figures 22 and 23 clearly show that the efficiency decreases with increasing cube size. This is not surprising because the analysis phase of Chapter V stated that the estimator $MST_i^1(\nu)$ is not scalable due to the global communications. It is reasonable to assume that the

efficiency will continue to decrease beyond a cube size of three although this cannot be tested with AFIT's eight node hypercube.

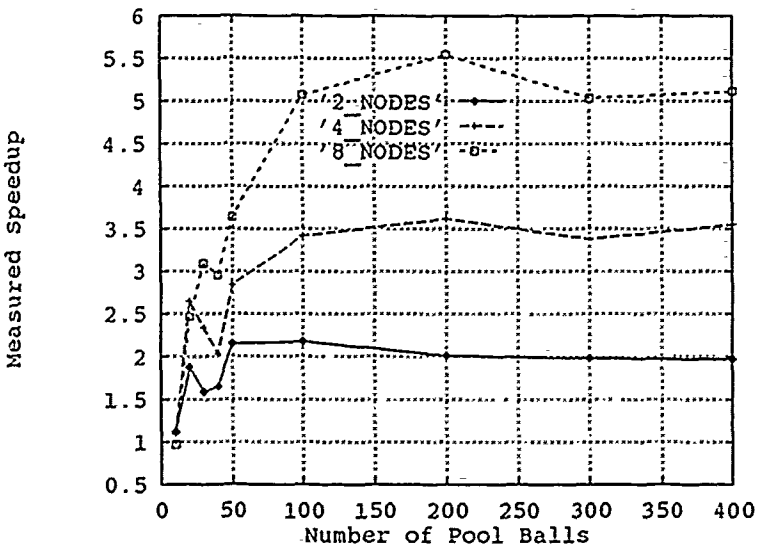


Figure 20. Speedup Curves as a Funtion of Load Factor

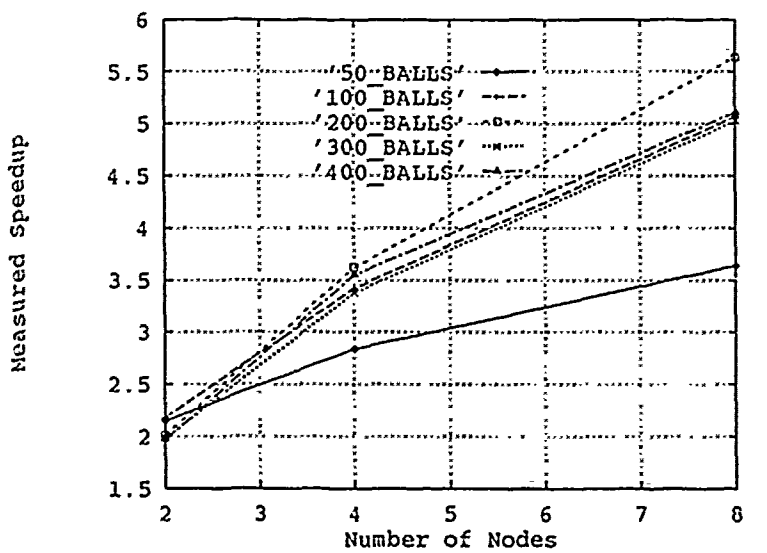


Figure 21. Speedup Curves as a Function of Cube Size

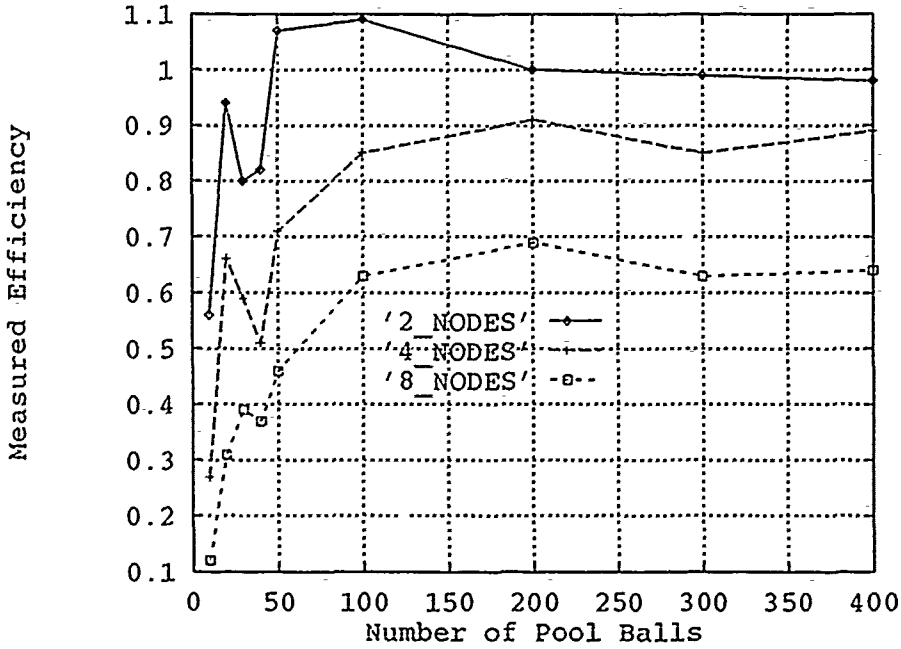


Figure 22. Efficiency Curves as a Function of Load Factor

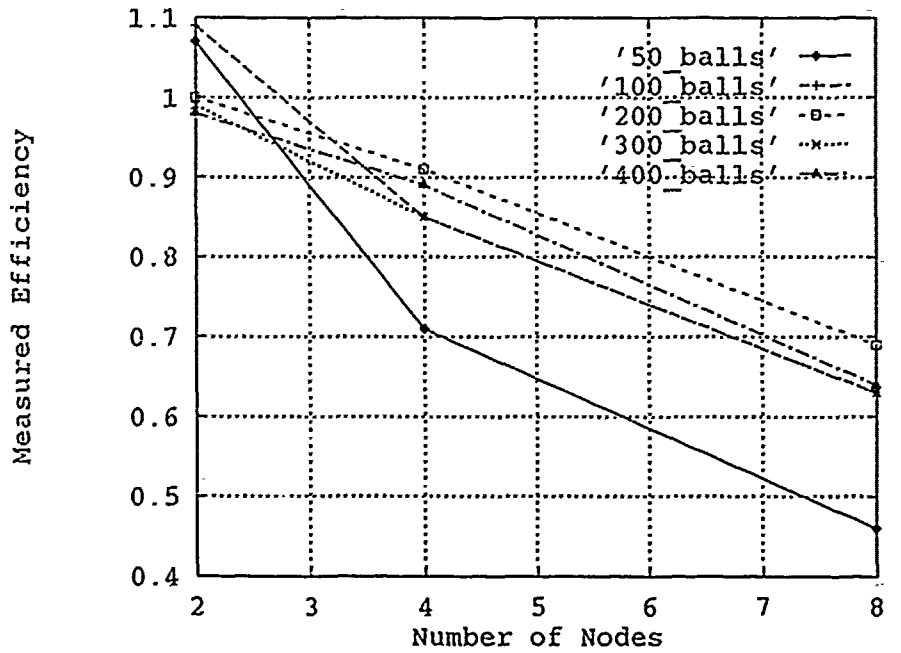


Figure 23. Efficiency Curves as a Function of Cube Size

Analysis of the data for optimum sectoring reveals an interesting trend. After 50 pool balls, the trend appears to establish a relationship between the optimum sectoring for all nodes and the number of pool balls per sector (density). The curves are shown as Fig 24. In fact, these curves were used to tailor the test plan for the 300 and 400 pool ball trials in an effort to reduce the time to complete the test runs. It was not surprising that the optimum sectoring is related to the pool ball density due to the tradeoffs discussed earlier in Chapter VI; however, another possibility exists. As the number of pool balls increases, the total number of events to process also increases. These events can be divided in two categories: events which are internal to a node and events which require communications. These correspond to the subsets {VERTICAL, HORIZONTAL, COLL} and {PARTITION, EXIT} respectively. Currently, the software keeps track of events by type (i.e. VERTICAL, HORIZONTAL, etc); however, with multiple sectors per node, not every PARTITION and EXIT event requires communications via message passing. Should a relationship exist between optimum sectoring and the ratio of internal event processing to external event processing, then a conclusion can be made regarding all simulations incorporating a two dimensional spatial partitioning scheme. Analysis of this relationship has not yet been accomplished.

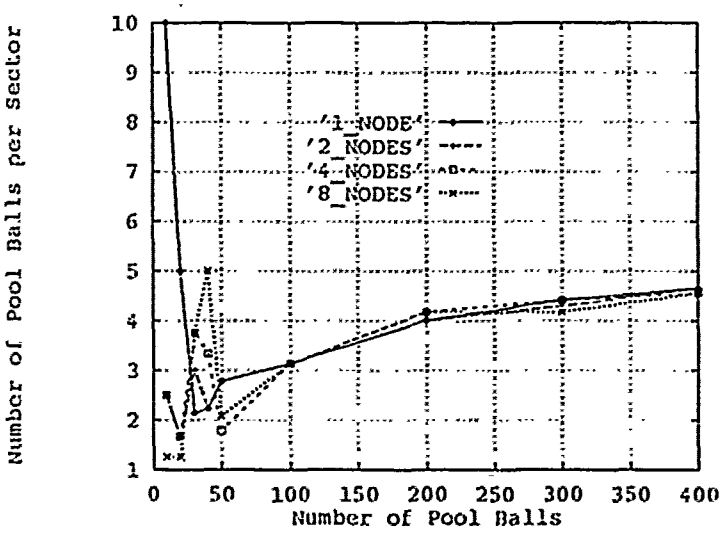


Figure 24. Density Curves as a Function of Load Factor

6.5 *Comparison of AFIT and Cal Tech Simulation Results*

Cal Tech developed the original pool balls simulation concept. Their research revolved around the Time Warp optimistic synchronization paradigm. The simulations developed by AFIT and Cal Tech are very similar, but not identical; therefore, comparisons are limited to general trends. The software designs for the two systems are different in several respects.

1. Each pool ball in AFIT's program has identical radius while Cal Tech's program allows for varying radii.
2. Each pool ball in AFIT's program has identical mass while Cal Tech's program allows for varying mass.
3. The AFIT software design establishes the entire table as a single object. Cal Tech's software design divides the four table cushions into sections, each of which is an object.
4. Cal Tech describes each pool ball (or puck) to be a separate object whereas the AFIT design defines a class of pool balls and a single ball manager object (3).
5. Cal Tech chose the more efficient $O(N)$ algorithm to manage a next event queue containing events. The AFIT simulation algorithm is $O(N^2)$ and avoids complex event list manipulation by storing on average only one event at a time.

The first four differences enumerated above should not affect the test results for either system in any appreciable manner. These differences represent implementation decisions. The implementation differences will probably result in different execution times but relative speedup measurements enable valid comparison exercises. Item five, however, represents a significant design difference. As the Intel hypercube is course grain, the $O(N^2)$ algorithm provides a better match between software and hardware. Thus, one would expect that an $O(N^2)$ simulation would result in superior speedup over an $O(N)$ algorithm, all else remaining the same. This highlights another potentially significant difference between the two simulations; that is, the hardware is not the same. Cal Tech used the JPL Mark III hypercube whereas AFIT used the Intel iPSC/2 hypercube. With differing granularities, the two machines will produce different speedup results even for the same software. During the course of this research, no subjective measures of granularity were found to adequately compare the two machines.

Cal Tech reported in the 1988 SCS Multiconference that speedup on eight nodes for 120 pool balls was approximately 3.2 (3:5). The simulation run time was not specified nor was the sectoring specified. There was no mention concerning optimal sectoring for the single node and 8 node configuration. For this thesis, a 120 pool ball scenario was generated for two seconds simulation time. The speedup on eight nodes was 4.97. The graph is shown as Fig 25. Cal Tech also published the speedup results using 160 pool balls. Their speedup was approximately 4.0 on eight nodes using 64 table sectors. From the reported test, it appears that Cal Tech did not use optimum table sectoring to calculate speedup. They chose instead to fix the sectoring for each of the cube sizes from one to 32 nodes. Test results using 160 pool balls were reported using 16, 32 and 64 sectors for which the 64 sector table produced the best of the three speedup results for all cube sizes. The AFIT simulation produced speedup of 5.40 for 160 pool balls on 8 nodes.

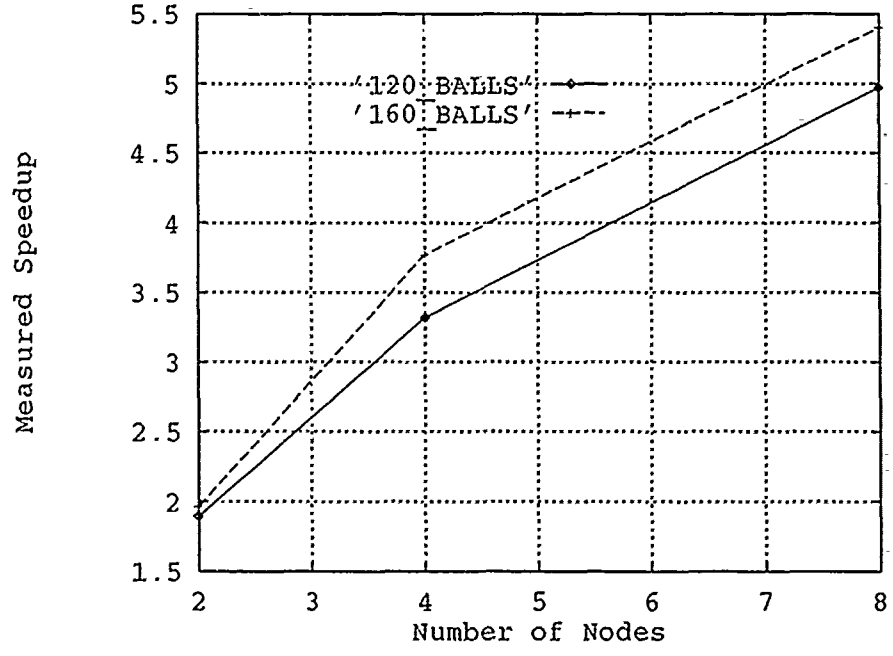


Figure 25. AFIT Speedup Curves for 120 & 160 Pool Balls

VII. Conclusions and Recommendations

7.1 Introduction

This chapter discusses the conclusions which can be made from the results of Chapter VI. These conclusions are conjectured to apply not only to the pool balls simulation but also to any simulation which incorporates a two or three dimensional space through which the objects of study move or pass. Examples include particle dynamics and battle field simulations.

7.2 Impact of Computational Load

Speedup is independent of the number of objects to be simulated for large N provided that optimum sectoring is used. The empirical data suggests that the speedup stabalizes at approximately $N = 100$

7.3 Impact of Spatial Partitioning

Sectoring the pool table results in a tradeoff between decreased search space and increased number of events to process. Each additional sector adds two incremental events to process for pool balls which must cross the additional sector. These events are not real events of interest and therefore represent overhead. Empirical data suggests that the optimal number of sectors increases with N, that it is dependant upon the ratio of pool balls to table sectors (density), and that the optimum number of sectors for any given size of N is independent of the cube size.

7.4 Determining the Optimal Number of Sectors

The empirical data shown in figure 24 suggests that the relationship between the optimum number of sectors and the number of pool balls is logarithmic. Using linear regression on an equation of the form $Y = a \log X + c$, where Y is the optimum number of sectors and X is the number of pool balls to be simulated, and minimizing the error of the model results in $a = 2.3$ and $c = -1.5$. The empirical data is plotted with the optimum sectoring estimate in figure 26.

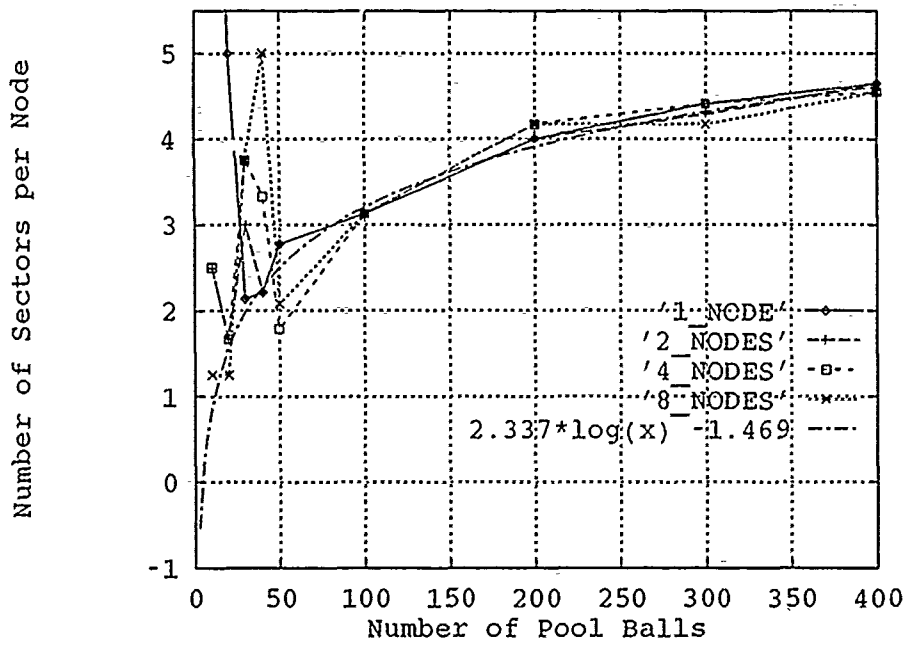


Figure 26. Curve Fitting to the Optimal Sector Data

7.5 Impact of the Conservative Paradigm upon Scalability

This thesis developed two formulations, $MST_i^1(\nu)$ and $MST_i^3(\nu)$, for calculating the minimum safe time. The estimator $MST_i^1(\nu)$ represents the tightest upper bound possible at the cost of global communications. The estimator $MST_i^3(\nu)$ complies with the constraints formally presented by Chandy and Misra. This approach uses less information to produce a lower estimate of the minimum safe time; however, communications time is a constant limited only to nearest neighbor communications. The approach favored by Chandy and Misra can result in an indeterminant number of iterations in which no sector can meet its minimum safe time. The additional information provided by global communications of the first scheme has been proved to guarantee that at least one sector can always meet its minimum safe time. The efficiency curves of figure 2 clearly illustrate the cost of global communications upon the parallel performance of the pool balls simulation. As the number of nodes increases, the efficiency generally decreases. Due to the global communications, $MST_i^1(\nu)$ is conjectured to yield superior speedup over the nearest neighbor scheme for small M only where M is at least four nodes. Due to the indeterminate number of idle iterations which can result with $MST_i^3(\nu)$, it is not possible to predict the value of

M for which the two estimators will produce equal speedup without empirical test data. The second estimator has not yet been implemented.

7.6 Conservative Versus Optimistic Paradigms

There are several differences between the pool balls simulation implemented by Cal Tech and AFIT. The two most significant differences are the overall algorithm time complexity and the hardware used to measure speedup. Cal Tech's simulation incorporated an $O(N)$ search algorithm while AFIT's simulation incorporated an $O(N^2)$ algorithm. Cal Tech used a JPL Mark III 32 node hypercube while AFIT uses an Intel iPSC/2 hypercube. There were no quantitative or qualitative measurements for hardware granularity for comparison. It is reasonable to assume that the measured speedup results for either simulation design would be different if run on different machines; therefore, an accurate comparison is not possible. Lin and Lawzowska performed an analytical study of the two paradigms and concluded that the optimistic approach is generally superior and that in the worst case, an optimistic approach cannot lag arbitrarily behind the conservative model. The speedup results presented in this thesis are approximately 35% higher on eight nodes than Cal Tech's reported speedup on eight nodes. While this does not disprove Lin and Lazowska's work, it does indicate that a conservative paradigm applied to a distributed discrete event simulation can produce significant speedup. This has important ramifications because an optimistic approach can require vast amounts of memory to execute. Chandy and Misra, on the other hand, have shown that their paradigm requires a bounded amount of memory and that the memory requirements are not more than for a sequential simulation. This thesis concludes that the conservative paradigm has many useful applications where the optimistic approach would otherwise exhaust memory. If designed properly, a conservative simulation can produce significant speedup.

7.7 Recommendations for Future Study

The concept of spatially partitioning the pool table for the pool balls simulation worked exceptionally well. This was due predominantly to the fact that the pool balls were uniformly distributed. A battlefield simulation is not guaranteed to have this advantage. Most of the computational load for a battlefield simulation occurs at the boundary between opposing forces. To ensure that as many sectors as possible have objects located within them, a dynamically defined sectoring scheme must be implemented. The pool balls concept is readily understood and is therefore easier to work with than complex battlefield

simulations. It is recommended that a dynamically assigned sectoring strategy can be created using the pool balls problem domain. The insights gained from this endeavor would apply to any non-uniformly distributed object model.

The design of the pool balls simulation intentionally incorporated an $O(N^2)$ algorithm to match the design with the granularity of the iPSC/2 hypercube. An $O(N)$ would require less overall execution time; therefore, it is recommended that the pool balls simulation be re-designed to accommodate the $O(N)$ algorithm. This allows further comparison with the pool balls simulation tested at Cal Tech. If the speedup of the $O(N)$ algorithm implemented using a conservative paradigm still outperforms the $O(N)$ algorithm using an optimistic paradigm, then the degree of confidence in the asserions stated in this thesis increase.

AFIT has been experimenting with a standard conservative synchronization package called SPECTRUM. This package was originally written at the University of Virginia and incorporates multiple filters for internodal communications. The main thrust of SPECTRUM is isolate machine dependant software in a low level implementation layer of software and to separate the synchronization protocal from the specific application. It is recommended that the pool balls software design be modified to interface to UVA's SPECTRUM package. This would standardize the pool balls simulation with other simulations produced at AFIT and would allow the software to be more easily ported to other distributed processing systems.

This thesis developed two minimum safe time estimators. The first estimator does not conform to Chandy-Misra's paradigm, but it is more efficient for small cube sizes. The second estimator does conform to Chandy-Misra's paradigm, but is less efficient and more scalable. This thesis implemented the pool balls simulation using the first estimator which is more efficient but less scalable. It is recommended that the pool balls simulation be redesigned with the second estimator for detailed performance comparison. This recommendation is based on the premise that massive parallelism is desired by the DoD to solve many of its complex battle simulations and large VHDL descriptions.

A two dimensional sectoring strategy is recommended for future investigation. Several large simulations, such as a battlefield simulation, are better suited to two dimensional sectoring due to the distribution of objects. By partitioning the domain in along two axes, the objects can be dispersed with a better chance of achieving superior load balancing. This conjecture can be tested with the pool balls problem.

Appendix A. *Software Listings*

A.1 Software Files

The software for the parallel implementation of the pool balls problem includes the following files:

- Initialize.c ⇒ Main Host Code
- Simdrive.c ⇒ Main Node Code
- ball_ADT.c
- table_ADT.c
- queue_ADT.c
- EventHandler.c
- Communicator.c
- clock.c
- event.c
- neq1.c
- pool_balls.c
- random.c
- cube.h
- event.h
- prototype.h
- structure.h

A.1.1 Functional Description

Initialize.c: This is the main program for the host. Its primary functions are to read in the command line arguments from the user, create the specified number of pool balls, create the pool table with the specified table dimensions, initialize all simulation parameters, place the pool balls into the appropriate table sectors, enforce Wieland's data replication for pool balls residing in border regions, and communicate this information to the individual nodes. The host program then enters a continuous loop until receiving a message from a node indicating that the simulation is over at which time the host kills all processes.

simdrive.c: This is the main program for the nodes. Its primary purpose is to enforce the high level loop construct for the simulation. The loop consists of:

- Determine the next event
- Determine the minimum safe time
- Schedule the next event(s)
- Execute the scheduled event(s)
- Enforce Data Replication

pool_balls.c: This is the main application code driver for the nodes. It has an initialization procedure which is called only once by the main code, simdrive.c. This procedure receives the command line arguments sent from the host, creates the pool table for each node, receives each pool ball sent from the host and inserts the pool balls into the appropriate sector, and initializes all other packages which require initialization. It is important to note that each node only receives the pool balls which are assigned to a sector which resides on that node. Last, the initialization program determines the first event(s) to be executed.

The program 'pool_balls.c' also has a procedure which is continuously called by the main node program which implements (via procedure calls) the funcions of executing an event, determining the next event, determining the minimum safe time, and scheduling the next event.

EventHandler.c: This file performs two basic tasks: it determines the next event possible for a pool ball which is passed to it, and it executes an event message which is passed to it.

Communicator.c: This file performs all of Wieland's data replication strategy. Each time a pool ball is moved, this file is invoked. The pool ball state information which is passed to the file is inspected. All replication and de-replication occurs here.

ball_ADT.c: This file is implemented as an abstract data type. It represents the ball object manager for each node. The manager can perform the following operations on a pool ball:

- Add

- Remove (without returning a pool ball)
- Get_and_Delete
- Get (without removing a pool ball)

Several other functions are available but do not change the state of the ball object manager. These operations include printing, error checking, counting, and searching.

clock.c: Manages the simulation clock.

event.c: Dynamically allocates and deallocates memory for events.

neq1.c: Manages the next event queue. Each node has one NEQ and each NEQ stores scheduled events. Each NEQ also has one dummy event to ensure that the queue is never empty. If the dummy event is popped, it must be re-inserted before the simulation can proceed.

queue_ADT.c: This file has two basic functions: it creates, manages and updates the candidate queues, and it enforces the synchronization protocol. It currently enforces the MST^1 estimator. Each sector has one candidate queue which stores candidate events. After each sector determines its minimum safe time, each candidate queue is inspected to determine if its candidate event is less than or equal to its minimum safe time. If it is, then the candidate event is popped off the queue and inserted into the node's next event queue. The event is now said to be scheduled. If the candidate event is greater than the minimum safe time, then the candidate event is removed and its memory is freed.

random.c: This file is a machine independant, pseudo-random number generator based on the works of Law and Kelton. It can return random numbers with the following distributions:

- uniform
- exponential
- normal
- logarithmic

table_ADT.c: This file is implemented as an abstract data type. The ADT represents the table sector manager. It not only creates the table and sectors, but it returns information about the table such as the table length, table width, and sector coordinates.

cube.h: This header file includes all cube specific information as 'define' statements in the C programming language.

event.h: This header file defines the data structure of an event type.

prototype.h: This header file prototypes all of the files used in the pool balls simulation.

structure.h: This header file defines all global structures such as the ball structure, linked list structure, and sector structure. It also defines several global variables such as the length of a pool ball radius, the value of π, the maximum allowable X and Y axis pool ball velocities and some key initialization parameters such as writing to dis and printing to screen.

A.2 Compiling Instructions

The makefile specifies how the various host and node files compile. The makefile is as follows:

```
host:       Initialize.o        random.o        ball_ADT.o    table_ADT.o

    cc -o   host Initialize.o   random.o        ball_ADT.o    table_ADT.o
            -lm -host


node:       simdrive.o          pool_balls.o    clock.o       neq1.o
   event.o              random.o        ball_ADT.o    table_ADT.o
            EventHandler.o      Communicator.o  queue_ADT.o

    cc -o   node simdrive.o     pool_balls.o    clock.o       neq1.o
            event.o             random.o        ball_ADT.o    table_ADT.o
            EventHandler.o      Communicator.o  queue_ADT.o
            -lm -node

Initialize.o:  Initialize.c  structure.h  cube.h prototype.h
    cc -c      Initialize.c

simdrive.o:    simdrive.c    event.h  cube.h  prototype.h  structure.h
    cc -c      simdrive.c

clock.o:       clock.c
    cc -c      clock.c
```

```
neq1.o:         neq1.c  event.h
    cc -c       neq1.c

event.o:        event.c  event.h
    cc -c       event.c

random.o:       random.c
    cc -c       random.c

pool_balls.o:   pool_balls.c  structure.h prototype.h
    cc -c       pool_balls.c

ball_ADT.o:     ball_ADT.c  structure.h prototype.h
    cc -c       ball_ADT.c

table_ADT.o:    table_ADT.c  structure.h
    cc -c       table_ADT.c

EventHandler.o: EventHandler.c  structure.h  event.h  prototype.h
    cc -c        EventHandler.c

queue_ADT.o:    queue_ADT.c  structure.h  event.h prototyp.h cube.h
    cc -c       queue_ADT.c

Communicator.o: Communicator.c  structure.h event.h prototype.h  cube.h
    cc -c       Communicator.c
```

Bibliography

1. Andrew Beard, Gary Lamont and others. "Compendium of Parallel Programs for the Intel iPSC Computers, Volume I, Version 1.5." Research Product of AFIT, Department of Electrical and Computer Engineering, June 1990.
2. Banks, Jerry and John S. Carson. *Discrete-Event System Simulation.* Englewood Clifs: Prentice-Hall, 1984.
3. Beckman, Brian. "Distributed Simulation and Time Warp - Part 1: Design of Colliding Pucks." *Proceedings of the SCS Multiconference on Distributed Simulation.* 56 - 60. San Diego: Simulation Councils, Inc, 1988.
4. Beckman, Brian and others. "The Status of the Time Warp Operating System," *ACM*, 7:738–744 (February 1988).
5. Chandy, K. M. and Jayadev Misra. "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Communications of the ACM*, *24*:198–206 (April 1981).
6. Chandy, K. Mani and Jayadev Misra. "Distributed Simulation: A Case Study in Design and Verification of Distribute Programs," *IEEE Transactions on Software Engineering*, *5*:440–452 (September 1979).
7. Cleary, John G. "Colliding Pucks Solved Using A Temporal Logic." *Proceedings of the SCS Multiconference on Distributed Simulation.* 219 - 224. San Diego: Simulation Councils, Inc, 1990.
8. Conklin, Darrell and others. "The Sharks World." *Proceedings of the SCS Multiconference on Distributed Simulation.* 157 - 160. San Diego: Simulation Councils, Inc, 1990.
9. DeCegama, Angel L. *The Technology of Parallel Processing.* Englewood Cliffs: Prentic Hall, 1989.
10. Green, Jim, Research Engineer. Telephone interview. Georgia Tech Research Institute, Redstone Arsenal, Al, 5 May 1991.
11. Hayes, John P. *Computer Architecture and Organization.* New York: McGraw Hill, 1988.
12. Hwang, Kai and Faye A. Briggs. *Computer Architecture and Parallel Processing.* New York: McGraw-Hill, 1984.
13. Jefferson, David. "Performance of the Colliding Pucks simulation on the Time Warp Operating Systems (Part 1: Asynchronous Behavior and Sectoring)." *Proceedings of the SCS Multiconference on Distributed Simulation.* 3 - 7. San Diego: Simulation Councils, Inc, 1989.
14. Law, Averill M. and W. David Kelton. *Simulation Modeling and Analysis.* New York: McGraw-Hill, Inc, 1991.

15. Lin, Yi-Bing and Edward D. Lazowska. "Optimality Considerations of 'Time Warp' Parallel Simulation." *Proceedings of the SCS Multiconference on Distributed Simulation*. 219 – 224. San Diego: Simulation Councils, Inc, 1990.

16. Lipton, Richard J. and David W. Mizell. "Time Warp vs Chandy-Misra: A Worst-Case Comparison." *Proceedings of the SCS Multiconference on Distributed Simulation*. 219 – 224. San Diego: Simulation Councils, Inc, 1990.

17. Lubachevsky, Boris D. "Simulating Colliding Rigid Disks in Parallel Using Bounded Lag without Time Warp." *Proceedings of the SCS Multiconference on Distributed Simulation*. 194 – 202. San Diego: Simulation Councils, Inc, 1990.

18. Misra, Jayadev. "Distributed Discrete-Event Simulation," *Computing Surveys*, *18*:39–65 (March 1986).

19. Pritsker, A. Alan. *Introduction to Simulation and SLAM*. West Lafayette: Halsted Press, 1986.

20. Schruben, Lee. "Modeling Systems Using Discrete Event Simulation." *Winter Simulation Conference Proceedings*. 101–108. Arlington: IEEE, 1983.

21. Stone, Harold S. *High-Performance Computer Architecture*. Reading: Addison-Wesley, 1990.

22. Wieland, Frederick and others. "An Empirical Study of Data Partitioning and Replication in Parallel Simulation." Future publication, June 1989.